Query Optimization for Database-Returning Queries

SIMON RINK, Saarland University, Saarland Informatics Campus, Germany
JENS DITTRICH, Saarland University, Saarland Informatics Campus, Germany

Recently, the novel concept of database-returning SQL queries (DRQs) was introduced. Instead of a single, (potentially) denormalized result table, DRQs return an entire subdatabase with a single SQL query. This subdatabase represents a subset of the original database, reduced to the relations, tuples, and attributes that contribute to the traditional join result. DRQs offer several benefits: they reduce network traffic in client-server settings, can lower memory requirements for materializing results, and significantly simplify querying hierarchical data. Currently, two state-of-the-art algorithms exist to compute DRQs: (1.) $\text{ResultDB}_{\text{Semi-Join}}$ builds upon Yannakakis' semi-join reduction algorithm by adding support for cyclic queries. (2.) $\text{ResultDB}_{\text{Decompose}}$ computes the standard single-table result and projects the result to the base tables to obtain the resulting subdatabase.

However, multiple issues can be identified with these algorithms. First, $\text{ResultDB}_{\text{Semi-Join}}$ employs simple heuristics to greedily solve the underlying enumeration problems, often leading to suboptimal query plans. Second, each algorithm performs best under different conditions, so it is up to the user to choose the appropriate one for a given scenario. In this paper, we address these two issues. We propose two enumeration algorithms for $\text{ResultDB}_{\text{Semi-Join}}$ to handle the *Root Node Enumeration Problem (RNEP)* and the *Tree Folding Enumeration Problem (TFEP)*. Further, we present a unified enumeration algorithm, $\text{TD}_{\text{ResultDB}}$, to automatically decide between plans generated by our new enumeration algorithms for $\text{ResultDB}_{\text{Semi-Join}}$ and $\text{ResultDB}_{\text{Decompose}}$.

Through a comprehensive experimental evaluation, we show that the enumeration time overhead introduced by our methods remains minimal. Furthermore, by effectively solving the RNEP and TFEP, we achieve up to a 6x speed-up in query execution time for $\text{ResultDB}_{\text{Semi-Join}}$, whereas $\text{TD}_{\text{ResultDB}}$ consistently selects the best available plans.

CCS Concepts: • **Information systems** → **Structured Query Language**; **Query optimization**.

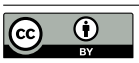
Additional Key Words and Phrases: query optimization, query processing, SQL, denormalization, subdatabase, Yannakakis, graph theory, result database

ACM Reference Format:
Simon Rink and Jens Dittrich. 2025. Query Optimization for Database-Returning Queries. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 353 (December 2025), 26 pages. https://doi.org/10.1145/3769818

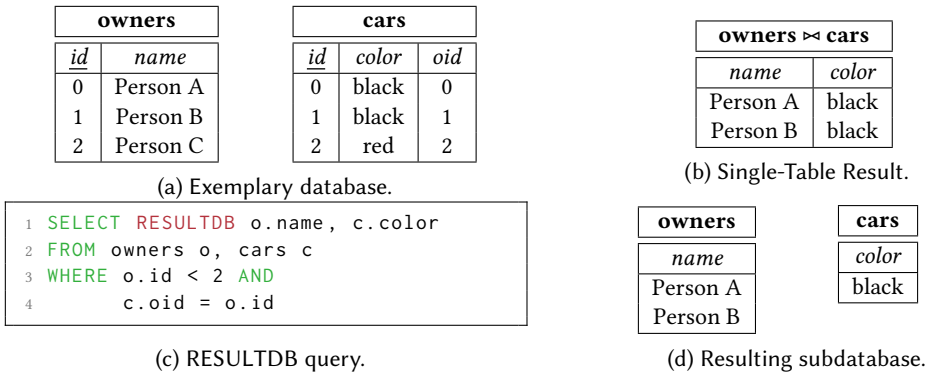
1 Introduction

In their recent work, Nix and Dittrich [26] introduced the `RESULTDB` operator, implementing the thrilling concept of a database-returning query (DRQ). DRQs return a reduction of the original database, which contains only the relations, tuples, and attributes that would take part in the traditional join result. An example is given in Figure 1, which shows that the returned subdatabase shown in Figure 1d is just a reduction of the original database from Figure 1a. Note that Figure 1d contains no redundancies, unlike the single-table result from Figure 1b. This becomes especially important for N:M queries with a heavy workload, substantially reducing output sizes, which can

Authors' Contact Information: Simon Rink, Saarland University, Saarland Informatics Campus, Germany, simon.rink@bigdata.uni-saarland.de; Jens Dittrich, Saarland University, Saarland Informatics Campus, Germany, jens.dittrich@bigdata.uni-saarland.de.



ACM 2836-6573/2025/12-ART353
https://doi.org/10.1145/3769818



(a) Exemplary database.

(b) Single-Table Result.

(c) RESULTDB query.

(d) Resulting subdatabase.

Fig. 1. RESULTDB Example.

significantly reduce network traffic in diverse client-server setups, such as distributed DBMSs [6]. Note that one would need to *post-join* the result database on the client side, i.e., join all returned tables, in case a denormalized result was required. In that case, required join keys would also need to be transmitted, independently of whether the keys are part of the projection. Beyond this application, DRQs also offer additional benefits, such as potential memory overhead reductions for large intermediate results, preventing relational information loss, and simplifying the process of querying hierarchical data [26].

Naturally, this idea has far-reaching implications for query processing and optimization. So far, the state-of-the-art [26] to compute DRQs consists of two algorithms for the same problem: $\text{ResultDB}_{\text{Semi-Join}}$ and $\text{ResultDB}_{\text{Decompose}}$.

$\text{ResultDB}_{\text{Semi-Join}}$ utilizes a modified version of Yannakakis' semi-join reducer [33]. It interprets the query graph as a tree, initially applying semi-join reductions from the leaf nodes to a designated root node, followed by another round of semi-join reductions from the root back to the leaves. This ensures that all relations contain only the tuples required for the regular join. However, these tree reductions are only feasible if the graph is acyclic [5]. To still be able to deal with cyclic graphs, the concept of *folding* is introduced, where a cyclic query is joined until it has an acyclic join graph, allowing for the execution of Yannakakis' base algorithm again. Therefore, $\text{ResultDB}_{\text{Semi-Join}}$ modifies the established query processing, and by that, introduces new optimization problems in the context of DRQs, specifically the *Root Node Enumeration Problem (RNEP)* and the *Tree Folding Enumeration Problem (TFEP)* [26]. The RNEP deals with the optimal root node choice for semi-join reductions, given that the root heavily influences the overall runtime due to the tree shape's dependency on the root. For example, the different trees shown in Figure 2 could result in vastly different query execution times. The goal of the TFEP, on the other hand, is to find the most efficient way to transform a cyclic graph into an acyclic one. For example, multiple ways exist to resolve the cycle in Figure 3a.

$\text{ResultDB}_{\text{Decompose}}$ differs from $\text{ResultDB}_{\text{Semi-Join}}$ by computing the regular single-table results, followed by a projection on all base tables, a process coined *decomposing*. For this reason, its optimization solely depends on state-of-the-art single-table optimizers [12, 16, 24]. In [26], $\text{ResultDB}_{\text{Decompose}}$ was used as a naive baseline for evaluating $\text{ResultDB}_{\text{Semi-Join}}$ against, however, it proved to be highly efficient for queries with low redundancies.

Problem Statement. Two major issues can be identified with the state-of-the-art. First, $\textit{ResultDB}_{\textit{Semi-Join}}$ uses for both the RNEP and the TFEP a simple heuristic selecting nodes with the highest number of neighbors, and by that, does not utilize any enumeration. In the case of the RNEP, these nodes stem from the projection set only. Naturally, the created join plans can

Table 1. Table of Acronyms

Acronym	Definition
DRQ	Database-Returning Query
RNEP	Root Node Enumeration Problem
TFEP	Tree Folding Enumeration Problem
BEP	Block Enumeration Problem
BEP Set	Block Enumeration Problem Set
$\mathcal{FP}$	Folding Problem Set
CC	Connected Component
DFS	Depth-First Search
TVC	Two-Vertex Cut
SC	Solution Class for TD_{Fold}
CCP	Connected-Complement Pair
GHD	Generalized Hypertree Decomposition

be arbitrarily bad, raising the need for an enumeration-based solution. Second, $\text{ResultDB}_{\text{Semi-Join}}$ performs well when there is high redundancy in large result sets, whereas $\text{ResultDB}_{\text{Decompose}}$ excels under opposite conditions. However, both methods exist in isolation, i.e., there is no algorithm deciding which method to use for a query, leaving the decision up to the user. To the best of our knowledge, both issues remain unaddressed in the context of DRQs.

Contributions. This work contributes the following to the optimization of DRQs:

(1) A unified top-down enumeration algorithm for DRQs ($\text{TD}_{\text{ResultDB}}$) that evaluates whether to use plans generated by our new enumeration algorithms for $\text{ResultDB}_{\text{Semi-Join}}$ or $\text{ResultDB}_{\text{Decompose}}$. (Section 3)
(2) A new top-down enumeration algorithm (TD_{Root}) utilizing dynamic programming to solve the RNEP. (Section 4)
(3) A new top-down enumeration algorithm (TD_{Fold}) approximating optimal solutions for the TFEP. (Section 5 and Section 6)
(4) An extensive evaluation using an implementation in the state-of-the-art query execution engine mu*t*able [17] to show the potential of our contributions compared to the existing baseline for DRQs [26] as well as state-of-the-art heuristics based on Generalized Hypertree Decompositions [1, 32]. (Section 8)

Related work is discussed in Section 7.

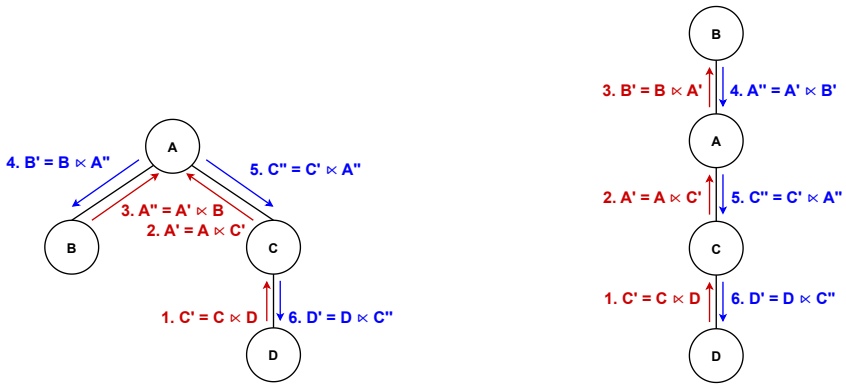
2 Preliminaries

This section summarizes the two algorithms $\text{ResultDB}_{\text{Semi-Join}}$ and $\text{ResultDB}_{\text{Decompose}}$ [26]. Both compute the result of a DRQ, i.e., a subdatabase only containing the base relations, tuples, and attributes contributing to the regular single-table query result.

2.1 $\text{ResultDB}_{\text{Semi-Join}}$

We will first present $\text{ResultDB}_{\text{Semi-Join}}$ for acyclic queries, followed by its adaptation for cyclic queries. It is important to mention the different definitions of acyclicity used in [26] and this paper:

Definition 2.1 (α-Acyclicity and JG-Acyclicity). Let Q be a query, and let $G(Q) = (V, E)$ be the join graph of Q, where V is the set of relations and E is the set of join edges. Q is called α-acyclic if it can be transformed into an equivalent query Q' where $G(Q')$ is a tree [11]. By contrast, Q is called *JG*-acyclic if $G(Q)$ is a tree [26].



(a) Root node A (b) Root node B

Fig. 2. Exemplary $\text{ResultDB}_{\text{Semi-Join}}$ execution for two different root nodes of the same join graph. Red arrows indicate bottom-up semi-joins, whereas blue arrows indicate top-down semi-joins.

We adapt the notion from [26], referring to acyclicity as *JG*-acyclicity. Note that an *JG*-acyclic query is always α-acyclic, but not vice versa.

2.1.1 Acyclic Queries. As shown in Figure 2, $\text{ResultDB}_{\text{Semi-Join}}$ works in three different phases for acyclic queries:

P1: Pick a root node based on a heuristic.
P2: Perform bottom-up semi-joins from the leaves to the root.
P3: Perform top-down semi-joins from the root to the leaves. This phase can stop early once all nodes in the projection set have been fully reduced.

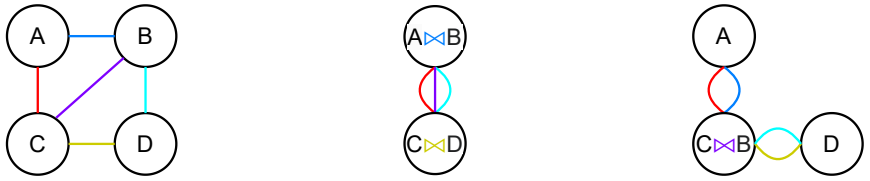
For instance, in Figure 2a, we start by reducing C with D to obtain the reduced node C', which is then used to reduce A to obtain A'. Afterward, A' is reduced by B to obtain the fully reduced A''. However, given that only the root is fully reduced at this point, the top-down pass is required to also fully reduce the remaining relations. During this process, we differentiate between two kinds of semi-join orders: vertical and horizontal.

The *vertical order* refers to the order in which relations of different depths in the graph are reduced. For example, in Figure 2a, Step 1 must precede Step 2. This order is fixed by the root node. On the other hand, the *horizontal order* refers to the sequence in which parents are reduced by their children. Unlike the prior order, this order is not determined by the root and can be chosen freely. For example, for correctness, it does not matter whether Steps 2 or 3 are performed first. Note that in Figure 2b, the horizontal order is fixed, as every node has only at most one parent or child. Thus, the choice of the root node also impacts the possibilities regarding the horizontal order, motivating the *Root Node Enumeration Problem*:

Definition 2.2 (Root Node Enumeration Problem). Let $G = (V, E)$ be an acyclic join graph, and let C be a cost function. Find the best root node $v \in V$ for $\text{ResultDB}_{\text{Semi-Join}}$ according to C.

The state-of-the-art selects the node from the projection set with the highest number of neighbors as the root. The quality of this choice is completely random, as no enumeration is involved in this process. Our solution, TD_{Root}, will implement a dedicated cost function, together with the corresponding enumeration approach based on dynamic programming.

2.1.2 Cyclic Queries. Yannakakis' base algorithm is not feasible for queries which are not α-acyclic [5, 26]. To mitigate this, the concept of *tree folding* was introduced in [26]. The idea is that certain relations are *folded* (i.e., joined) together, thus merging the corresponding vertices and edges of the



(a) Cycle. (b) 2-Join Solution. (c) 1-Join Solution.

Fig. 3. A cyclic join graph, together with ways to fold a cycle such that the resulting graph is acyclic.

join graph, effectively removing all cycles such that Yannakakis' semi-join reducer can be applied again. We call a set of nodes that are folded together a fold. For example, Figure 3b shows one way to solve the cycle presented in Figure 3a by folding A and B, as well as C and D. An alternative solution, folding only C and B, is shown in Figure 3c. Note that after the semi-join reduction has been applied, we need to decompose these folds into the corresponding base relations again. For that, the algorithm tracks what attributes belong to which base relation. The decomposing step can become computationally expensive with highly redundant data, as shown in Section 8. This motivates the need to minimize the computational cost caused by the folding as much as possible. For that, the *Tree Folding Enumeration Problem* is introduced:

Definition 2.3 (Tree Folding Enumeration Problem). Let Q be a query with a cyclic join graph $G = (V, E)$ and let C be a cost function. A subset $\mathcal{S} \subseteq 2^V$ is called a solution for G, when all $S \in \mathcal{S}$ are mutually disjoint, and when folding all elements of each $S \in \mathcal{S}$ will resolve all cycles in G. $\mathcal{S}$ is called optimal w.r.t C if $\mathcal{S}$ has the lowest cost according to C out of all possible solutions.

For example, Figure 3c is the result of applying the solution $\{\{A\}, \{C, B\}, \{D\}\}$ to Figure 3a. If a join graph is cyclic, the state-of-the-art approach joins nodes with the highest number of neighbors, as they are more likely to be part of a cycle. This approach has two issues. First, there is no guarantee that the joined relations are part of a cycle. Second, this does not consider the plan quality.
In this paper, we will present a solution, TD_{Fold}, tackling these issues. Different ways of how to solve the cycle via folding are enumerated, and in the end, the best solution found is chosen.

2.2 $\text{ResultDB}_{\text{Decompose}}$

This algorithm has the following approach: the regular single-table result is computed, which is then projected (or decomposed) to the required base relations and attributes. Normally, when computing a single-table result, semi-join reducers tend to increase the runtime tremendously compared to usual query processing [7, 26]. However, due to the need to eliminate duplicates, the final decomposition step becomes particularly costly in scenarios with many redundant tuples. This makes a case for $\text{ResultDB}_{\text{Semi-Join}}$, especially when optimized, which can shine in these scenarios.

3 $\text{TD}_{\text{ResultDB}}$ - Find the Best DRQ Plan

The primary goal of our contribution $\text{TD}_{\text{ResultDB}}$ is to answer the difficult question whether the optimized variant of $\text{ResultDB}_{\text{Semi-Join}}$ or $\text{ResultDB}_{\text{Decompose}}$ yield lower overall costs. For that, cost estimations of both algorithms are required. To determine optimized cost estimates and query plans for $\text{ResultDB}_{\text{Semi-Join}}$, we will first compute the best folding strategy according to TD_{Fold} and use the resulting graph to find the optimal root node according to TD_{Root}. The costs of both phases are then simply added to obtain the costs of $\text{ResultDB}_{\text{Semi-Join}}$. On the other hand, cost estimations for $\text{ResultDB}_{\text{Decompose}}$ simply involve estimating the costs of creating and decomposing the single fold containing all relations. Section 6 will discuss the cost estimations required for folds, which can simply be reused here. Naturally, subplans and costs can be shared between

enumerations for $\text{ResultDB}_{\text{Decompose}}$ and $\text{ResultDB}_{\text{Semi-Join}}$. Therefore, the following sections focus on the cost estimation and enumeration strategies for $\text{ResultDB}_{\text{Semi-Join}}$. Given that TD_{Fold} will require knowledge of TD_{Root}, we will start by introducing TD_{Root}.

4 TD_{Root} - Find the Best Root Node

This section introduces our new enumeration algorithm to optimize acyclic query processing in $\text{ResultDB}_{\text{Semi-Join}}$ by addressing the RNEP.

4.1 Intuition and Core Idea

While the root node choice does not affect the correctness of the results [5, 33], it significantly impacts the algorithm's efficiency in terms of memory and CPU usage. To enhance efficiency, we prioritize horizontal semi-joins with higher selectivity during Phase 2 of $\text{ResultDB}_{\text{Semi-Join}}$. This approach reduces the number of tuples in parent nodes as early as possible, yielding a more efficient overall execution.

Importantly, this selectivity-based horizontal order also introduces a deterministic execution sequence. This allows a cost estimation of a complete run for a fixed root node. Given the roots' impact on the overall runtime, it makes sense to enumerate each possible root node to choose the best one.

Existing methods for single-table results [32], to the best of our knowledge, simulate the whole tree traversals from both phases to estimate the resulting costs for each root. However, this approach leads to unnecessary computations, given that some parts of the executions (and thus their costs) will stay the same for different root nodes. For example, as seen in Figure 2, the overall structure of the tree heavily changes between the different root nodes. However, there are two elements that both have in common. First, in Step 1, C is reduced by the original D. Second, in Step 6, D is reduced by the maximally reduced C. Therefore, we can observe that the semi-join reductions below node A, i.e., in the subtree with root C will *not* change between the presented tree structures, both in Phases 2 and 3, implying that we should utilize dynamic programming to reuse the costs that would result from these reductions. To identify these situations, we can make use of the following lemma:

Lemma 4.1. Let $G = (V, E)$ be a connected, acyclic join graph, let T_{r_1}, T_{r_2} be trees of G with roots $r_1, r_2 \in V$, let $u \in V$ where $u \notin \{r_1, r_2\}$, and let T_u be a subtree of T_{r_1} and T_{r_2} with root u. Further, assume that the selectivity-based horizontal semi-join order is used in T_u. If u has the same parent in both T_{r_1} and T_{r_2}, then the semi-joins happening in T_u are the same in T_{r_1} and T_{r_2}

Proof. Proof via induction over u.

Base Case: u is a leaf in T_{r_1}. As u is a single node, the corresponding subtree only consists of one node. Therefore, no semi-joins happen in T_u, meaning the semi-joins in T_u are the same for both T_{r_1} and T_{r_2}.

Induction Hypothesis. For a subtree T_u of T_{r_1} and T_{r_2}, the semi-joins happening in T_u are the same for both T_{r_1} and T_{r_2}, if u has the same parent in both T_{r_1} and T_{r_2}.

Induction Step. u is an internal node in T_{r_1}. Let v be the parent of u in both T_{r_1} and T_{r_2}. Given that u always has v as parent, the children C_u of u are the same in both T_{r_1} and T_{r_2}. From our induction hypothesis, we know that the semi-joins happening in the subtrees T_c induced by all $c \in C_u$ will stay the same since u is their parent in both T_{r_1} and T_{r_2}. Given that the selectivity-based order yields a fixed horizontal order in T_u, the children of u will reduce u in a fixed order, regardless of the root. After u has been reduced with all its children, we are left with u'. u' is then used to reduce v. In Phase 3, we then reduce u' with the maximally reduced v to obtain u''. Afterward, u'' is used to reduce its children C_u in the same horizontal order as in Phase 2. The top-down semi-joins

happening in each T_c for all $c \in C_u$ will again be the same according to our induction hypothesis. Concluding, the semi-joins happening in T_u are the same for both T_{r_1} and T_{r_2}. □

In other words, the costs occurring in a subtree of G are uniquely identified by the root of the subtree, and the root's parent. The idea of the algorithm is then to traverse all nodes $v \in G$ and compute the costs of the subtree rooted at v, while using dynamic programming to reuse the costs of subtrees that have already been evaluated.

Algorithm 1 Find the best root node in G.

```
1: function TD_ROOT(G)
2:     (best_root, best_costs, pt) = (NULL, ∞, PlanTable())
3:     for root ∈ G.nodes do                                  ▷ Traverse all nodes
4:         root_costs = SUBTREE_COSTS(G, NULL, root, pt)
5:         if root_costs < best_costs then
6:             (best_root, best_costs) = (root, root_costs)   ▷ Update best root
7:     return Plan(best_root, best_costs)
```

Algorithm 2 Estimate the costs of a subtree.

```
1: function SUBTREE_COSTS(G, parent, root, pt)
2:     if parent ≠ NULL and (parent, root) in pt then
3:         return pt[parent, root]
4:     costs = 0
5:     for child ∈ selectivity_order(G, parent, root) do      ▷ Bottom-up phase
6:         costs += SUBTREE_COSTS(G, root, child, pt)
7:         costs += semi_join_costs(root, child)
8:         root = estimate_reduction(root, child)             ▷ Reduce root with child
9:     if parent ≠ NULL then                                  ▷ Top-down phase
10:        root = estimate_full_reduction(root)               ▷ Reduce root with parent
11:    for child ∈ selectivity_order(G, parent, root) do
12:        if reduction_required(child) then
13:            costs += semi_join_costs(child, root)
14:    if parent ≠ NULL then
15:        pt[parent, root] = costs
16:    return costs
```

4.2 The Algorithm

Building on the previous subsection's intuition, we now formally present our algorithm TD_{Root} in Algorithm 1. The function takes the current join graph G as input. In line 2, we first define the required variables and data structures. The plan table pt maps subtrees, identified by (parent, child) pairs, to costs. In line 3, we traverse each possible root node, estimate its costs using our new cost function (line 4), and update the best root found thus far in lines 5-6. We then return the final plan in line 7.

Algorithm 2 defines the cost estimation for the subtree rooted at a given node. Before discussing the actual algorithm, we first define some auxiliary functions:

- **selectivity_order(G, p, r)**: Iterates over the children of r in the order they should be traversed. The current parent of r, p, will be ignored. It is precomputed once for each node and stored for reuse. The precomputation requires $O(|V|^2 \log |V|)$ time at worst. It utilizes predetermined cardinality estimations and assumes selectivity independence. Each returned *child* is reduced by all its descendants. The iteration requires $O(|V|)$ time.
- **semi_join_costs(r, s)**: Estimates the costs of reducing r with s via a semi-join. Takes into account the current amount of tuples in r and s and requires $O(1)$ time.
- **estimate_reduction(r, s)**: Returns the state (i.e. remaining tuples) of node r after being reduced with s and requires $O(1)$ time.

- **reduction_required(r)**: Checks whether any node in the subtree starting with r is in the projection set, and thus needs to be reduced in the top-down phase. The check requires $O(1)$ time and utilizes precomputed results, that require $O(|V|)$ time once.
- **estimate_full_reduction(r)**: Returns the state of r after being fully reduced and requires $O(1)$ time.

Algorithm 2 takes the join graph G, the *root* of the subtree, the *parent* of *root* in G, and *pt* as input, while returning the costs occurring in the subtree. In lines 2-3, it starts by checking whether the current (*parent*, *root*) pair already exists in the table and returns the respective costs if that is the case. In line 5, we traverse each child in the selectivity order to account for the bottom-up semi-join costs and the recursive child costs. During the traversal, we start with a recursive call to Algorithm 2 (line 6), where the current *root* takes the role of the new *parent* and the *child* the role of the new *root*. Note that this call will return the costs from both the bottom-up and top-down phases. Line 7 adds the estimation of the semi-join costs between the root and the child, where the utilized child node is assumed to be reduced by all its descendents. Afterward, line 8 updates the current *root* by estimating its reduction with the *child*. This leaves the costs of the third phase to be explored. Here, we start in lines 9-10 by fully reducing the *root* with the *parent* (if a parent exists). Afterward, we again traverse each child in line 11 and compute the costs to reduce them (lines 12-13). A reduction is only required, when the child or any of its descendants is in the projection set. Afterward, the overall costs for the respective subtree are final. Therefore, we can update our plan table and return our results.

4.3 Time Complexity Analysis

In the following, a node visitation refers to any instance where a node is iterated by selectivity_order(G, p, r) (both bottom-up and top-down), or when evaluated as root. During a call to Algorithm 2, we visit each node at most once, and each visitation requires $O(1)$ time, thus Algorithm 2 runs in $O(|V|)$ time. TD_{Root} evaluates Algorithm 2 $O(|V|)$ times, meaning lines 3-6 require $O(|V|^2)$ time. Due to the required precomputations, the overall worst-case complexity is $O(|V|^2 \log |V|)$ time. Note that several graph structures offer much better performance, depending on how well they can profit from dynamic programming.

5 TD_{Fold} - Find the Best Folding Strategy

Section 4 discussed optimizing queries on tree-shaped join graphs. However, this excludes the important class of cyclic queries [7, 8], for which we must first solve the TFEP in order to apply Yannakakis' base algorithm. To this end, we propose the TD_{Fold} algorithm, which has the primary objective of identifying cost-effective strategies for eliminating cycles through folding, and consists of two phases:

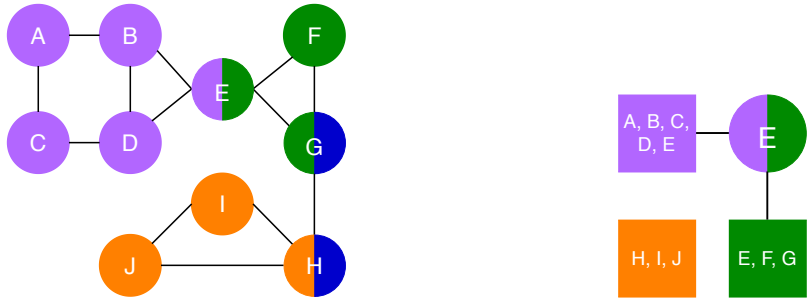
Phase 1: Create a so-called *folding problem set* $\mathcal{FP}$ of enumeration problems for each cycle within the graph. This functionality will be implemented by **create_folding_problem_set**(G).

Phase 2: Enumerate different solutions for each enumeration problem, and estimate the cost of each solution based on a new cost function to choose the best solution. This functionality will be implemented by **enumerate_folding_problem_set**(G, $\mathcal{FP}$, PT), where PT is a plan table.

Afterward, the chosen solution will be applied to G. This section will focus on the first phase, i.e., the implementation of **create_folding_problem_set**(G), whereas the second phase will be discussed in Section 6.

5.1 General Idea

The main goal of TD_{Fold} is to optimize queries with large N-M joins by minimizing redundancies. It does so by reducing the number of relations per fold, achieved by only considering joins within



(a) A cyclic join graph. Each distinct color represents a block.

(b) The block-cut forest for the given graph. Note that this is not a join graph.

Fig. 4. A cyclic graph, and the corresponding block-cut forest.

cycles. Still, within a cycle, various folding strategies with different relations per fold are explored. For that, we define sets of enumeration problems, where each enumeration problem is tied to a cycle in the join graph. The sets should satisfy the following properties:

P1: Within each enumeration problem, we are able to find at least one solution to the TFEP of the subgraph induced by the corresponding cycle.
P2: All enumeration problems are mutually disjoint.

These properties ensure that different cycles can be solved independently. For example, in Figure 4a, you can see a join graph with three different cycles, colored in orange, purple, and green. One example for an enumeration problem set for these cycles would be $\Big\{\{H, I, J\}, \{A, B, C, D, E\}, \{F, G\}\Big\}$. Within each of these mutually disjoint enumeration problems, we can find at least one solution to resolve the corresponding cycle. Having independent enumeration problems is highly desirable, as this simplifies the enumeration, avoids unnecessary joins between different cycles, and enables parallelization in query engines with multi-threading [23].

However, ensuring disjoint enumeration problems while guaranteeing a solution for the cycle of each enumeration problem is challenging due to overlapping cycles. The next sections address how to handle this.

5.2 Independent Enumeration Problems

In the previous subsection, we motivated why we want to create disjoint enumeration problems for each cycle. In order to do that, we first have to identify all cycles. While various methods exist [30, 31] for finding cycles, we opt for *blocks* [19], as they naturally partition the join graph into distinct cycles.

Definition 5.1 (Block). Let $G = (V, E)$ be a join graph. A connected subgraph $G' = (V', E') \subseteq G$ is called *biconnected* if there is no vertex $v' \in V'$ such that removing v' disconnects G'. G' is called a *block* if it is a maximal biconnected subgraph, i.e., there is no other biconnected subgraph $G'' = (V'', E'') \subseteq G$ with $V' \subset V''$. In this paper, a block B is solely defined by its vertices for simplicity reasons, i.e., $B = V'$.

For example, in Figure 4a, each block is illustrated using a distinct color. However, not every block corresponds to a cycle; only those with at least three vertices form a cycle [19]. Intuitively, since removing any node from the block does not disconnect it, there must be at least two distinct paths between any pair of nodes, implying the presence of a cycle. For example, in Figure 4a, every color except blue represents a cyclic block. We denote the set of blocks with three vertices by $\mathcal{B}_G$. Consequently, identifying the cycles of a graph is equivalent to finding $\mathcal{B}_G$ [21, 29]. Now, to

motivate our idea for **create_folding_problem_set**(G), as a simplification, we want to create a single set $\mathcal{P}_{\mathcal{B}_G}$ which contains a so-called *block enumeration problem (BEP)* for each block $B \in \mathcal{B}_G$, where a BEP is defined as follows:

Definition 5.2 (Block Enumeration Problem (BEP)). Let $B \in \mathcal{B}_G$ be a block. A set of vertices $P \subseteq B$ is called a BEP of B, if there is a subset $\mathcal{S} \subseteq 2^P$, such that $\mathcal{S}$ is a solution to the TFEP of the induced graph $G[B]$, i.e., a subgraph of G limited to the nodes of B.

For example, $P' = \{A, B, C, D\}$ is a BEP for the purple block in Figure 4a, since $\mathcal{S}' = \big\{\{A, B, C, D\}\big\}$ is a solution to the TFEP of the purple subgraph. However, $P'' = \{A, B, C\}$ is not a BEP because even folding all relations in P'' would not resolve the cycle in the purple subgraph. Therefore, using a BEP ensures property P1 of Section 5.1. The BEP P of a block B will later be used to enumerate different solutions $\mathcal{S} \subseteq 2^P$ for $G[B]$. Following property P2, we want each $P \in \mathcal{P}_{\mathcal{B}_G}$ to be disjoint. If $\mathcal{P}_{\mathcal{B}_G}$ adheres to both properties P1 and P2, we formally define it as a BEP set:

Definition 5.3 (BEP Set). Let G be a join graph, and let $\mathcal{B} \subseteq \mathcal{B}_G$ be a set of blocks. Then $\mathcal{P}_{\mathcal{B}}$ is called a BEP set of $\mathcal{B}$, if it contains a BEP for each B in $\mathcal{B}_G$, and all $P \in \mathcal{P}_{\mathcal{B}}$ are pairwise disjoint.

Unfortunately, we cannot simply set the BEP of each block equal to itself, as this would conflict with property P2, given the presence of so-called cut vertices, which are a special case of vertex cuts [19]:

Definition 5.4 (Vertex Cut). A subset $V' \subset V$ of a connected graph $G = (V, E)$ is called a vertex cut if removing V' from G disconnects G. If $|V'| = 1$, the single node from V' is called a *cut vertex* (sometimes referred to as *articulation point*), whereas for $|V'| = 2$, V' is called a *two-vertex cut*.

For instance, E is a cut vertex in Figure 4a as removing it will disconnect the remaining graph. To solve the problem of intersecting BEPs, we want to assign each cut vertex to exactly one $P \in \mathcal{P}_{\mathcal{B}_G}$, such that $\mathcal{P}_{\mathcal{B}_G}$ is a BEP set. For example, $\Big\{\{H, I, J\}, \{A, B, C, D, E\}, \{F, G\}\Big\}$ and $\Big\{\{H, I, J\}, \{A, B, C, D\}, \{E, F, G\}\Big\}$ would be BEP sets for $\mathcal{B}_G$. However, looking at these assignments, we can already see that our simplification to keep one BEP set per graph is suboptimal, due to two reasons. First, some BEPs are completely independent from other BEPs, like the BEP for the orange block, given that the orange block does not intersect with any other block. Second, multiple BEP sets can be created for the same set of blocks, requiring an enumeration to decide which one to use. Therefore, what we want to do is to create BEP sets for sets of intersecting blocks, i.e., the following set $\mathcal{FP}$, which we coin a *folding problem set*:

$$\Big\{\overbrace{\Big\{\underbrace{\{H, I, J\}}_{\text{BEP}}\Big\}}^{\text{BEP Set}}, \overbrace{\Big\{\underbrace{\{A, B, C, D, E\}}_{\text{BEP}}, \underbrace{\{F, G\}}_{\text{BEP}}\Big\}}^{\text{BEP Set}}, \overbrace{\Big\{\underbrace{\{A, B, C, D\}}_{\text{BEP}}, \underbrace{\{E, F, G\}}_{\text{BEP}}\Big\}}^{\text{BEP Set}}\Big\}$$

This leaves us with the question of how to create $\mathcal{FP}$ during **create_folding_problem_set**(G).

5.3 Creating BEP Sets

The previous section introduced the concept of BEP sets, which consist of disjoint BEPs. Specifically, we motivated that we aim to create BEP sets for sets of intersecting blocks. Therefore, this section will introduce our method for this.

In a first step, we want to identify the sets of connected blocks. To achieve this, we utilize so-called *block-cut forests*, adapted from *block-cut trees* [19]:

Definition 5.5 (Block-Cut Forest). Let $G = (V, E)$ be a connected join graph, and let $C \subseteq V$ be the cut vertices of G which are part of at least two blocks $B \in \mathcal{B}_G$. Then the block-cut forest BC_G of G is defined as an undirected graph (V_{BC}, E_{BC}), where $V_{BC} = \mathcal{B}_G \cup C$ and $(U, v) \in E_{BC}$ iff $U \in \mathcal{B}_G$, $v \in C$ and $v \in U$.

The forest consists of nodes for both blocks and cut vertices shared by at least two blocks. A cut vertex node is connected to a block node if it's part of that block in the join graph. Figure 4b illustrates this with three block nodes and one shared cut vertex node, E. A forest is needed (rather than a single tree) because we only consider blocks with at least three vertices, which may form multiple connected components (CCs) in the block cut forest. For example, the orange block node in Figure 4b is disconnected from the remaining forest. Therefore, these CCs correspond to the block sets we want to form separate BEP sets for. This raises the question: how do we create a BEP set for a CC of the block cut forest?

Each BEP set must satisfy properties P1 and P2. Property P2 is met by assigning each cut vertex to exactly one BEP. To ensure P1, we rely on the fact that for any block B and any node $v \in B$, the set $\{B \setminus v\}$ is always a solution for $G[B]$, as folding the set yields two nodes, resolving the cycle in B [26]. For example, $\{\{F, G\}\}$ is a solution for the green block in Figure 4a. This guarantees that $\{P\}$ is always a solution for a BEP P of B as long as $|P| \geq |B| - 1$.

While $\{P\}$ isn't necessarily the only solution, if $P \neq B$, however, a solution $\mathcal{S}$, where $|\mathcal{S}| \geq 2$ may not exist. For instance, $P = \{F, G\}$ has only one solution in Figure 4a. Further, solutions may still exist after losing multiple cut vertices, but we ignore these assignments since they do not guarantee a possible solution. Therefore, within a BEP set, we assign each cut vertex to only one BEP and ensure each BEP loses access to at most one cut vertex. To create a BEP set for a CC in the block cut forest, we follow the approach below.

Given the acyclicity of the CC [19], we treat the CC as a tree and choose a block node R from the CC as a root. We then set the BEP of R equal to R, whereas the BEP of all other blocks of that CC loses access to its cut vertex parent in the tree. This will yield a BEP set, as each block obtains a BEP, each cut vertex is assigned to the BEP of exactly one block (its parent in the tree), and the BEP of each block loses access to at most one cut vertex (again its parent in the tree). For example, when choosing $R = \{A, B, C, D, E\}$ as root in Figure 4b, then $\{\{A, B, C, D, E\}, \{F, G\}\}$ would be the resulting BEP set, as the root node got access to the whole block, whereas the green block lost access to its' parent in tree, E. Afterward, we repeat this procedure for all possible block root nodes in the CC. Performing these assignments for all CCs will result in the folding problem set $\mathcal{FP}$, which we can express using the following formula:

Definition 5.6 (Folding Problem Set Formula). Let $G = (V, E)$ be a join graph, let $BC_G = (V_{BC}, E_{BC})$ be the block-cut forest of G, let $\mathcal{C} \subseteq 2^{V_{BC}}$ be the set of CCs of BC_G, and let $Bl(CC)$ be the set of block nodes in a $CC \in \mathcal{C}$. Further, when a $CC \in \mathcal{C}$ is interpreted as a tree with root $R \in Bl(CC)$, then the set only containing the parent of $B \in Bl(CC)$ within that tree can be referenced by $\text{parent}(CC, B, R)$. The folding problem set $\mathcal{FP}$ is then given by

$$\bigcup_{CC \in \mathcal{C}} \left\{ \bigcup_{R \in Bl(CC)} \left\{ \left\{ R \right\} \cup \left(\bigcup_{\substack{B \in Bl(CC), \\ B \neq R}} \{ B \setminus \text{parent}(CC, B, R) \} \right) \right\} \right\}$$

Algorithmically, that means, that during **create_folding_problem_set**(G), we must first compute the blocks and cut vertices using the method from [21]. Then, we use them to construct the block-cut forest. Both can be done in $O(|V| + |B_G|)$. In order to assign cut vertices for a given root within a CC, we can use a depth-first search (DFS) [30], which can also be done in $O(|B_G|)$ time. In

the worst case, we have $|B_G| - 1$ possible assignments with our approach, and since $|B_G| \leq |V|$, we know that the assignment can be computed in $O(|V|^2)$ time. After the assignment has been completed, we know that every set in $\mathcal{FP}$ only contains BEP sets, shown by the following lemma.

Lemma 5.7. Let $G = (V, E)$ be a cyclic, connected join graph, let $BC_G = (V_{BC}, E_{BC})$ be the block-cut forest of G, and let $\mathcal{C} \subseteq 2^{V_{BC}}$ be the set of CCs in BC_G. Further, for a $CC \in \mathcal{C}$, let $Bl(CC)$ denote the block nodes in CC, and let $FP(CC)$ be the set of BEP sets associated with $Bl(CC)$. Then for the folding problem set $\mathcal{FP}$ it holds that, $\forall CC \in \mathcal{C}.\forall \mathcal{P} \in FP(CC).\mathcal{P}$ is a BEP set for $Bl(CC)$.

Proof. Obviously, for each $CC \in \mathcal{C}$, a set of BEP sets is created for $Bl(CC)$. That means, we need to show that for any arbitrary CC, it holds that any $\mathcal{P} \in FP(CC)$ is a BEP set for $Bl(CC)$. Let $R \in Bl(CC)$ denote the chosen root to create $\mathcal{P}$, and let $P(B) \in \mathcal{P}$ denote the BEP associated with a block $B \in Bl(CC)$. For each $B \in Bl(CC)$, there will be exactly one set $P \in \mathcal{P}$, such that $P = P(B)$, as each block is traversed in Defintion 5.6 exactly once. Lastly, for each $B \in Bl(CC)$, $P(B)$ is a BEP, because $|P(B)| \geq |B| - 1$, since $P(B)$ only has lost access to its parent in BC_G and by that, a possible solution to the TFEP of $G[B]$ would be $\{P(B)\}$. □

At this stage, we have demonstrated how to create different BEP sets for each CC in the block cut forest. Therefore, we are left with enumerating solutions within $\mathcal{FP}$ in order to find the best solution (w.r.t. the given cost function) for each CC in the block cut forest. The next section will discuss our algorithm for that.

6 Enumerating Folding Problem Set Solutions

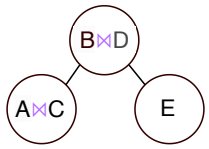
In the previous section, we discussed how to create a folding problem set $\mathcal{FP}$ containing different BEP sets for each CC in the block cut forest of a join graph. Building on this, in this section we will discuss the algorithm used to enumerate the solutions of $\mathcal{FP}$: **enumerate_folding_problem_set**(G, $\mathcal{FP}$, PT). We start by presenting its general idea. Starting with the solution of a single BEP P, let $S(P) \subseteq 2^P$ be the best solution of P, and let $C(P)$ be the costs of $S(P)$. Given that BEP sets consist of independent BEPs (property P2), we can simply combine all their solutions. That means that the costs $C(\mathcal{P})$ of a BEP set $\mathcal{P}$ are defined as $\sum_{P \in \mathcal{P}} C(P)$, and the solution $S(\mathcal{P})$ of $\mathcal{P}$ as $\bigcup_{P \in \mathcal{P}} S(P)$. Then, the algorithm will identify for each $FP \in \mathcal{FP}$ the BEP set $\mathcal{P}' \in FP$ with the lowest costs, and set the costs $C(FP)$ and solution $S(FP)$ of FP equal to $C(\mathcal{P}')$ and $S(\mathcal{P}')$. Now, we know that each $FP \in \mathcal{FP}$ stems from a different CC of the block-cut forest, and by that, they are mutually independent, so we can simply combine their solutions again. The costs and solution of $\mathcal{FP}$ will then be equal to $\sum_{FP \in \mathcal{FP}} C(FP)$ and $\bigcup_{FP \in \mathcal{FP}} S(FP)$. For instance, a solution for the folding problem set from Section 5.2 would be $\Big\{\{H, I, J\}, \{A, B, C\}, \{D, E\}, \{F, G\}\Big\}$

This leaves one remaining question: how can we find the best solution and costs for a given BEP? We start by introducing our cost model in the next subsection.

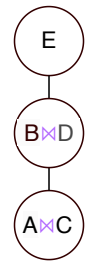
6.1 The Cost Model for BEP Solutions

Algorithm 3 Exemplary Hash-Based Semi Join Heuristic.

```
1: function SEMI_JOIN_HEURISTIC(G, P, F)
2:     (curr_costs, root) = (0, F.node)
3:     for child ∈ selectivity_order(G, P, NULL, root) do          ▷ Bottom-Up
4:         curr_costs += root.number_of_tuples
5:         root = estimate_reduction(root, child)
6:     for child ∈ selectivity_order(G, P, NULL, root) do          ▷ Top-Down
7:         if reduction_required(child) then
8:             curr_costs += root.number_of_tuples
9:     return curr_costs
```



(a) Using the folded node as root.



(b) Using a different node as root.

Fig. 5. Two possible subtrees for the node created by folding $\{B, D\}$ from Figure 4a.

Since a solution $\mathcal{S} \subseteq 2^P$ of a BEP P always consists of disjoint folds, we define a cost model at the level of individual folds. The total cost of $\mathcal{S}$ can be then be computed as the sum of the fold costs. Consequently, our cost function to estimate the query processing costs of a fold $F \in 2^P$ consists of three component cost functions:

(1) The **costs of the join order** to join all relations in F.
(2) The **costs of decomposing** the fold into its base relations after the semi-join reductions have been completed.
(3) The **costs of the semi-join reductions** in the final acyclic tree involving the node created by folding F.

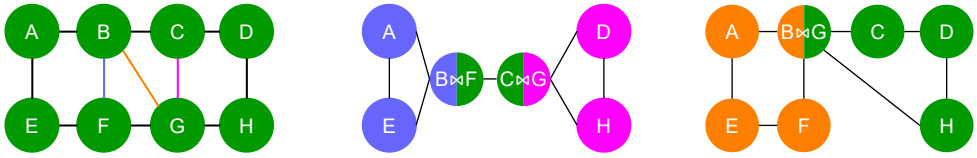
To estimate the costs for (1), we can simply reuse well-established cost functions (e.g. C_{Out} [9]). To estimate the costs for (2), we examine the decomposing operation: for each tuple in the maximally reduced fold (i.e., after the semi-join reductions), we need to project it onto the attributes of each relation of the fold that is part of the final projection. When a fold consists of only a single relation, no decomposition is required, so the associated costs can simply be set to 0. Therefore, to estimate the decomposing costs of a fold F, we use the cost function $C_{\text{Decompose}}(F)$:

Definition 6.1 ($C_{Decompose}$). Let $G = (V, E)$ be a join graph, let $k \in \mathbb{N}$ be an implementation-specific factor, let $F \subseteq V$ be a fold storing n tuples after being fully reduced, and let $F' \subseteq F$ be the set of relations participating in the final join result. Then:

$$C_{\text{Decompose}}(F) = \begin{cases} 0, & |F| = 1 \\ n \cdot k \cdot |F'| & \text{otherwise} \end{cases}$$

However, estimating (3) is especially challenging, as efficient cost computations are critical. The difficulty lies in the fact that during TD_{Fold}, the parent nodes of the folded node F_{node} (i.e., the node created by folding F) are not yet known. Ideally, we would evaluate the cost for all possible parents of F_{node}, but this is computationally expensive. To address this, we use a heuristic: we assume F_{node} will become the root of the final tree and estimate the number of tuples from F_{node} that will be processed during the semi-join reduction. A specific example for hash-based semi-joins is shown in Algorithm 3.

This algorithm closely follows Algorithm 2, but only estimates the processed tuples at the root. The function **selectivity_order**(G, P, NULL, r) is modified to include the BEP P, using the optimal horizontal order of P_{node} across all folds under the assumption of independent selectivities. Neighbors of P that are not neighbors of F are ignored, while neighbors in $P \setminus F$ are added dynamically. Additionally, if a set of connected neighbors N belongs to the same cyclic block, it is replaced by a single node representing the join of N to maintain acyclicity. For instance, in Figure 5a, the subtree considered for the fold $\{B, D\}$ of Figure 4a is shown. Here, neighbors of the purple block are replaced with their join. These replacements do not affect the estimation, as the



(a) A graph with TVCs. (b) Using both TVCs. (c) Using only one TVC.

Fig. 6. An example containing TVCs, and two resulting folded graphs when two or only one TVC(s) are folded. Colored vertices represent blocks, whereas colored edges represents two-vertex cuts within blocks.

heuristic focuses only on by which neighbors a folded node is reduced. Overall, Algorithm 3 runs in $O(|V|)$ time per fold, with a preliminary sorting step per BEP requiring $O(|V| \log |V|)$ time. While this approach underestimates the true costs, e.g., when F_{node} is not the root, as in Figure 5b, it helps to avoid particularly inefficient plans, especially in cases of large differences in (intermediate) result sizes. Note that Algorithm 3 is only used within TD_{Fold}, i.e., when deciding for the best folds, and is discarded in favor of the actual semi-join costs when creating the final tree with TD_{Root}.

Therefore, we define our cost model C_{Fold} to estimate the costs of a fold F as:

Definition 6.2 (C_{Fold}). Let $G = (V, E)$ be a join graph, let $P \subseteq V$ be a BEP, let $F \subseteq P$ be a fold, and let $C_{\text{Join}}(F)$ be any cost function determining the cost of joining all relations of F. $C_{\text{Fold}}(G, P, F)$ is then defined as:

$$C_{\text{Join}}(F) + C_{\text{Decompose}}(F) + \text{semi\_join\_heuristic}(G, P, F)$$

Concluding, different solutions for a BEP can now be evaluated using the new cost model C_{Fold}. Note that, given a join graph $G = (V, E)$, the costs of evaluating $\text{ResultDB}_{\text{Decompose}}$ is simply equal to $C_{\text{Fold}}(G, V, V)$, as Algorithm 3 just returns 0 in that case. This leaves us with the discussion of how to enumerate the different solutions of a given BEP.

6.2 Enumerating BEP Solutions

Algorithm 4 Greedily apply TVCs and return the resulting graph.

```
1:  function GET_GREEDY_JOIN_GRAPH(G, P, B)
2:      G' = G[B]
3:      tvcs = get_two_vertex_cuts(G, P)
4:      tvcs.sort_by_number_of_intersections_ascendingly()
5:      applied_folds = ∅
6:      for tvc ∈ tvcs do                                  ▷ Traverse all TVCs
7:          if tvc & applied_folds ≠ ∅ then                ▷ Only apply non-intersecting TVCs
8:              continue
9:          G' = G'.fold(tvc)
10:         applied_folds = tvc ∪ applied_folds
11:     return G'                                          ▷ Return the folded subgraph
```

The previous subsection demonstrated how to evaluate the costs of a given fold, and by that, the entire solution of a BEP. Therefore, this section will discuss how we plan to enumerate various solutions of a BEP. There is already a significant body of research on various join enumeration algorithms [12, 16, 24], so our approach aims to leverage these existing methods to take advantage of their strengths and efficiencies. In the following, we will discuss three different solution classes (SCs) that we intend to enumerate for each BEP P of a block B.

SC_{1F}: Folding until one fold is left, which will trivially always resolve the cycle. For that, we need to evaluate $C_{\text{Fold}}(G, P, P)$ once. To enumerate all possible join orders, we will utilize the well-established bottom-up join enumerator DP_{CCP}.

Algorithm 5 Enumerate different solutions for a BEP.

```
1: function ENUMERATE_BEP(G, P, B, PT)
2:     if PT.has_folding_plan(P) then
3:         return PT.folding_plan[P]
4:     if !PT.has_join_plan(P) then
5:         dp_ccp(G, P, PT, c_out)
6:     ▷ Join all relations in the BEP
7:     PT.update_folding_plan(P, c_fold(G, P, P), {P})
8:     ▷ Join until two folds are left if possible
9:     if P = B then
10:        for CCPs (P1, P2) ∈ 2^P × 2^P where P1 ∪ P2 = P do
11:            costs = c_fold(G, P, P1) + c_fold(G, P, P2)
12:            PT.update_folding_plan(P, costs, {P1, P2})
13:    ▷ Try to create smaller blocks with TVCs
14:    G' = (V', E') = get_greedy_join_graph(G, P, B)
15:    if G' == G[B] then
16:        return PT[P].folding_plan                    ▷ No TVCs were found
17:    FP'' = create_folding_problem_set(G', B − P)
18:    ▷ Recursively enumerate new blocks
19:    (folds, costs) = enumerate_folding_problem_set(G', FP'', PT)
20:    ▷ Not all nodes in V' are part of a BEP in FP'', but part of the solution
21:    for v ∈ (V' \ (B − P)) that are not an element of any BEP of FP'' do
22:        costs += c_fold(G', {v}, {v})
23:        folds.add({v})
24:    PT.update_folding_plan(P, costs, translate(folds, G))
25:    return PT[P].folding_plan
```

SC_{2F}: Folding until two folds are left, which will always resolve the cycle [26]. This is only guaranteed to be possible when $B = P$, as discussed in Section 5.2, so we will only consider SC_{2F} when $B = P$. To obtain these final two folds, we need to enumerate pairs of subproblems $(P_1, P_2) \in 2^P \times 2^P$, where $P_1 \cup P_2 = P$, $P_1 \cap P_2 = \emptyset$, and both P_1 and P_2 are connected. In other words, we seek to identify all top-level connected complement pairs (CCPs) [24] of P. To accomplish this, we can utilize the top-down join enumeration algorithm $TD_{MinCutBranch}$ [12], which allows us to avoid re-enumerating all possible bottom-up join orders. Therefore, we need to evaluate C_{Fold} for both pair elements of each top-level CCP.

SC_{TVC}: Using two-vertex cuts (TVCs) to divide blocks into smaller blocks, which can then be recursively enumerated.

Since SC_{TVC} is more complex than the other two, we explain it in more detail. Before showing how TVCs help split blocks, we briefly outline why this works. A TVC is a node pair whose removal disconnects the graph. For example, $\{B, F\}$, $\{B, G\}$, and $\{C, G\}$ are TVCs in Figure 6a. Joining a TVC within a block effectively merges its connectivity into a single cut vertex, producing smaller blocks and requiring fewer joins to resolve the cycle than SC_{1F} and SC_{2F} alone.

Lemma 6.3. Let $G = (V, E)$ be a connected join graph, and let $C \subset V$ be a vertex cut in G, with $|C| \geq 2$. Then the node $f' \in V'$ resulting from folding C is a cut vertex in the folded join graph $G' = (V', E')$.

Proof. From the definition of a vertex cut, we know that removing C splits G into a set of disjoint connected subgraphs $\mathcal{S}'$. Let E_C be the edges connecting C with $V \setminus C$. During the folding, we copy G to G', remove C and E_C from G', and add a new node f' to V', that represents the fold of C. Finally, we add $E_f = \{\{f', v'\} | v' \text{ is an unfolded node in } G', \text{ that connected } V \setminus C \text{ with } C \text{ in } G\}$ to E'. It follows that each subgraph $S' \in \mathcal{S}'$ is connected to f' only via edges from E_f, thus all $S \in \mathcal{S}$ in G' are now connected via f' only. Therefore, removing f would split G' into $\mathcal{S}'$ again. Thus, f' is a cut vertex in G'. □

As shown in Figure 6b, joining $\{B, F\}$ and $\{C, G\}$ creates three new blocks, enabling recursive enumeration. To avoid the exponential cost ($O(2^n)$ for n TVCs) of evaluating all combinations of which TVCs to apply, we greedily apply as many TVCs as possible to reduce the number of relations per fold. However, care is needed: folding $\{B, G\}$ instead, as shown in Figure 6c, results in only two blocks, as $\{B, G\}$ intersects with both other TVCs and alters their connectivity when joined. We therefore prioritize TVCs with minimal overlap and only join non-intersecting TVCs simultaneously.

This greedy folding strategy is given by Algorithm 4. Here, we first copy the induced subgraph $G[B]$ to the new graph G' in line 2. Afterward, we compute all two vertex cuts limited to P in line 3 using the strategy described in [15, 20]. Afterwards, in line 4, **sort_by_number_of_intersections_ascendingly()** will sort all TVCs in ascending order according to their pairwise intersections, which will be done using Counting Sort. Afterward, we greedily apply non-intersecting TVCs to G' in lines 6-10, prioritizing TVCs appearing earlier in the TVC list. Finally, we return the folded graph G'. Algorithm 4 runs in $O(|V| + |E|)$ time [15, 20].

Next, we want to utilize **create_folding_problem_set**$(G', B - P)$ to construct a new folding problem set $\mathcal{FP}''$ for G'. The function now takes an additional input: $B - P$, i.e., the set only containing the removed cut vertex from P, v, if any. After creating an intermediate set $\mathcal{FP}'$ according to Definition 5.6, the function additionally removes v from all BEPs in $\mathcal{FP}'$. Any BEP sets of $\mathcal{FP}'$ that violate property P1 as a result are discarded. This is necessary to maintain isolation with other BEPs of G, as Definition 5.6 only considers the assignments local to G', and because of that, ignores that P might not have access to v. Note that there will always be at least one BEP set per CC in G', as you can always choose a block B' containing v as root for Definition 5.6. As B' then still only looses one vertex, the corresponding BEP still fulfills property P1, whereas others BEPs are unaffected.

Crucially, join and decomposition costs must still be estimated with respect to the vertices from the original graph G, not the folded graph G'. For instance, for the BEP $\{C \bowtie G, D, H\}$ in Figure 6b, the enumeration happens on G', possibly yielding $\{\{C \bowtie G, D\}, \{H\}\}$ as a potential solution. Since $\{C \bowtie G, D\}$ corresponds to $\{C, G, D\}$ in Figure 6a, the join and decomposition cost estimation must use the original fold, with the cost model handling this translation automatically. Finally, we will utilize a plan table PT to store various plans and costs. Let P' be a BEP of a block B' of any (potentially folded) graph G'. For P', it will store the best solution to resolve the cycle in $G'[B']$, as well the corresponding costs (PT.folding_plan$[P']$). Further, for subproblems $P'' \subseteq P'$, it will store the best join order and costs (PT.join_plan$[P'']$). To allow reusability between different folded graphs, these will be stored and looked up using the fold corresponding to P'' in the original graph G. Lastly, for P'', the value of $C_{\text{Fold}}(G', P', P'')$ will be stored. For convenience, calling PT.c_fold(G', P', P'') will return the corresponding costs, if already evaluated, and compute and store them first otherwise. With all these ideas in place, we will now present Algorithm 5.

The algorithm gets a join graph G, a block B, the BEP P of B, and a plan table PT as input. In lines 2 and 3, we first check whether P has been evaluated already, and if so, return the corresponding folding plan entry. If not, we check whether the best join order for P has been computed already, and if not, compute it using the well-known DP_{CCP} [24] algorithm. Afterward, in line 7, we estimate the costs of SC_{1F}, followed by the costs of SC_{2F} in lines 9-12. Finally, in lines 14-24, we apply SC_{TVC} as described above. In particular, we create a greedily folded (sub-)graph G' in line 14. If $G' \neq G[B]$, G' is utilized to create a new folding problem set $\mathcal{FP}''$ in line 17, which is recursively enumerated in line 19. Note that some folded TVCs might not belong to any blocks in G', thus we have to manually account for these in lines 21-23, as they still have to be folded to obtain a valid solution. Further, the folds contained in *folds* are based on the nodes in G', thus we need to translate them

back into equivalent folds of G, which is done by **translate**(*folds*, G) in line 24. Finally, we return the best folding entry from our plan table.

Theorem 6.4. Let $G = (V, E)$ be a cyclic join graph, let $B \subseteq V$ be a block of G, and let $P \subseteq B$ be a BEP of B. Then, the solution returned by Algorithm 5 is a solution for the TFEP of $G[B]$.

Proof Sketch. Proof via induction over $G[P]$.
Base Case: There is no TVC in $G[P]$. Then the solutions of SC_{1F} and, if $B = P$, SC_{2F} are a solution for the TFEP of $G[B]$ [26].
Induction Hypothesis. The solution returned by Algorithm 5 is a solution for the TFEP of $G[B]$.
Induction Step. There is at least one TVC in $G[P]$. Then, the solutions of SC_{1F} and SC_{2F} are still valid solutions for the TFEP of $G[B]$. That means, we need to show that SC_{TVC} produces a solution for the TFEP of $G[B]$. Let $G' = (V', E')$ be the folded graph created in line 14, and let $\mathcal{B}_{G'} \subseteq 2^{V'}$ denote the set of blocks in G'. Following that, we know from Lemma 5.7 that the intermediate set $\mathcal{FP}'$ produced by **create_folding_problem_set**(G', $B - P$) is a folding problem set for G'. Further, from Definition 5.6, we know that in the BEP sets of $\mathcal{FP}'$, there is one BEP P' for each $B' \in \mathcal{B}_{G'}$ where $P' = B'$. This means that removing $v \in B - P$ from P' will not change the fact that P' is a BEP. Therefore, $\mathcal{FP}''$ will contain at least one BEP set for each CC of $\mathcal{B}_{G'}$. Let U' denote the set of nodes $v' \in (V' \setminus (B - P))$, which are not an element of any BEP of the BEP sets of $\mathcal{FP}''$. Following the induction hypothesis, Algorithm 5 will return for each $B' \in \mathcal{B}_{G'}$ a solution for B', labelled as $S(B')$. Line 19 will then return the set $\mathcal{S}_{G'} = \bigcup_{B' \in \mathcal{B}_{G'}} S(B')$. Since $\mathcal{S}_{G'}$ contains a solution for each $B' \in \mathcal{B}_{G'}$, $\mathcal{S}_{G'}$ is a solution for G'. Let $\mathcal{S}_G = (\bigcup_{v' \in U'} \{\{v'\}\}) \cup \mathcal{S}_{G'}$. Since $\mathcal{S}_{G'}$ is a solution for G', we know that translated($\mathcal{S}_G, G$) is a solution for the TFEP of $G[B]$. □

It follows directly, that the solutions by **enumerate_folding_problem_set**(G, $\mathcal{FP}$, PT) are also a solution to the TFEP of G. Thus, we are left with the time complexity analysis of **enumerate_folding_problem_set**(G, $\mathcal{FP}$, PT), which we will show in the upcoming subsection.

6.3 Time Complexity Analysis

This section discusses the time complexity of **enumerate_folding_problem_set**(G, $\mathcal{FP}$, PT). For simplicity, we ignore SC_{TVC} and will focus on the enumeration of one block first. We know from Section 5, that for each block B, there might be multiple BEPs for B, namely $B_C + 1$, where B_C is the number of cut vertices of B. For simplicity, we assume $|V|$ as an upper bound of B_C, which will not change the overall complexity. Each of these BEPs must be solved *once* only due to the plan table. For a BEP P of B with $P = B$, we must enumerate all possible join orders, which can be done in $O(3^{|B|})$ time [24]. Further, for each top-level CCP (P_1, P_2), we additionally must evaluate C_{Fold} for both P_1 and P_2, which can be done in $O(2^{|B|}|V|)$ time [12], as C_{Fold} must be evaluated for each connected subset of B. For a BEP P' of B with $P' \neq B$, we do not have to enumerate anything anymore, as $C_{\text{Fold}}(P')$ is already known from the enumeration of P, thus can be estimated in $O(1)$ time. Looking at all blocks, the solution of all BEPs can be computed in $O(\sum_{B \in \mathcal{B}_G} 3^{|B|}) \leq O(3^{|V|})$ time. Afterward, we simply have to combine the costs and solutions of all BEP sets for each $FP \in \mathcal{FP}$, done in $O(|V|^2)$ time.

Now we also account for SC_{TVC}, again starting with considering a single block B. We need to consider $|B|$ initial subgraphs, one for each possible BEP of B, depending on which cut vertex has been removed. For each of these $|B|$ initial subgraphs, we might have to consider at most $|E_B|$ differently folded graphs, where E_B is the set of edges in B, as you could apply TVCs $|E_B|$ times in the worst case. Therefore, we have at most $|B||E_B|$ different subgraphs to consider per block. Creating all these folded graphs will require $O(|B||E_B| * (|V| + |E|))$ time. Afterward, we need to create a folding problem set for each of the resulting subgraphs, each requiring $O(|B|^2)$ time to create, thus overall $O(|E_B||B|^3)$. For each of these, we have to solve SC_{1F} and SC_{2F} again.

However, we no longer need to compute the join orders, and need to compute C_{Fold} at most $O(2^{|B|})$ times, requiring $O(2^{|B|}|V|)$ time in the worst case per block. Regardless, we must evaluate the costs of the best folds in SC_{2F} for each of the $|B||E_B|$ different subgraphs. The evaluation for one subgraph can be done in $O(2^{|B|})$ time, thus requiring $O(2^{|B|}|B||E_B|) \leq O(2^{|B|}|V||E|)$ for the entire block. Consequently, choosing the best assignment for each subgraph can be done in $O(|B|^3|E_B|)$. Therefore, looking at all blocks, we have $O(\sum_{B \in \mathcal{B}_G} 2^{|B|}|V||E|) \leq O(2^{|V|}|V||E|)$ additional overhead. Therefore, the overall complexity is given by $O(3^{|V|})$, the same complexity as DP_{CCP}.

7 Related Work

This section presents selected works that are closely related to our proposals.

7.1 Blocks and Cut Vertices

The concepts of blocks and cut vertices have been utilized for various purposes. For example, in [10], DeHaan and Tompa utilized cut vertices in their *Biconnection Tree* to ensure that only CCPs are enumerated in their top-down enumeration approach. Their biconnection tree resembles a block-cut tree but includes additional nodes that are not necessarily cut vertices. Further, the biconnection tree is used during the enumeration itself, whereas our work merely uses blocks and vertex cuts to prepare certain enumeration problems, which are then enumerated using other methods. Another interesting work stems from Mancini et al. [23], which uses blocks to create smaller enumeration problems, whose CCPs are then enumerated in a multi-threaded system that allows for a faster enumeration time for large queries. That multi-threaded approach would also be perfectly usable for our approach but is beyond the scope of this paper.

7.2 Full Reducer Problem

Naturally, the classical full reducer problem, where relations are first reduced before eventually being joined, is related to the DRQ problem. However, the most important difference is the fact that both problems have different objectives, and by that, approaches that are beneficial to one problem might not be beneficial to the other. For example, avoiding joins like we do might decrease the performance of full reducer algorithms [1, 32], however, adding joins like, e.g., in [32] might decrease the performance of DRQs due to the decomposing overhead. Still, certain techniques from the full reducer problem are related to our work, which we will discuss in the following.

Generalized Hypertree Decompositions (GHDs). Because Yannakakis' algorithm [33] is not applicable to cyclic queries, existing full reducer algorithms often make use of GHDs [1, 13, 14, 32] to transform cyclic queries into trees by putting connected vertices into so-called bags while adhering to certain properties to ensure the eventual acyclicity. All vertices within a bag are subsequently joined. However, existing state-of-the-art optimizers [1, 32] use the AGM bound [2] to minimize the *fractional hypertree width* of a given GHD, i.e., the highest *theoretical* number of tuples within any bag, ignoring the actual cardinalities of joins. Further, GHDs enumerate invalid decompositions that would not resolve the cycles and have to be filtered out, and also consider joins outside of cycles.

Diamond Hardened Joins. In [7], Birler et al. proposed their diamond-hardened join framework, which avoids the creation of large intermediate results by splitting up joins into multiple suboperators. By doing this, the work also mitigates the overhead caused by the index structures commonly used in *worst-case optimal joins* [25] and reduces the need of factorization methods [27, 28] in intermediate results. We believe a combination of our approach with the method described in [7] could further improve the efficient computation of DRQs, particularly during fold computation.

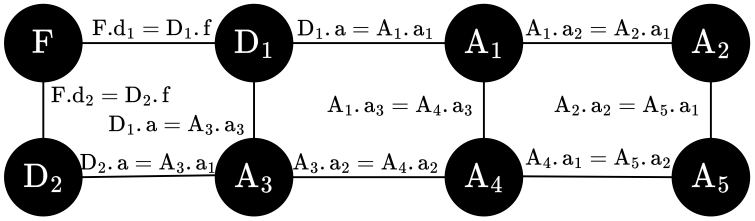


Fig. 7. The query for the synthetic dataset.

8 Evaluation

This section presents the results of our benchmarks. During the evaluation, we want to answer the following two questions regarding our new enumeration algorithms:

- **Q1:** How fast is the enumeration compared to the state-of-the-art for DRQs and state-of-the-art GHD heuristics? (Section 8.2)
- **Q2:** How good is the plan quality, i.e., the actual query execution time, compared to the state-of-the-art for DRQs and state-of-the-art GHD heuristics? (Section 8.3)

Note that early studies of DRQs were made in the context of distributed DBMSs in [26]. However, our optimization is context-agnostic, i.e., relevant for all possible use-cases of DRQs, as we focus on optimizing the computation of the DRQ itself. Concretely, the resulting subdatabase will always be the same as in [26], which means that measurements for compression ratios, transfer times, or post-join execution times would yield the same results as in [26]. Because of that, we will not conduct any experiments based on these metrics.

8.1 Experimental Setup

We start with our experimental setup.

Hardware & Software. We utilize a MacBook Pro with an M4 Max 16-Core processor, together with 48 GB of main memory. Further, the underlying OS is macOS Sequoia 15.1.

Database System. The algorithms described in Sections 4-6 are implemented into the state-of-the-art query execution engine mu*t*able [17]. At its core, mu*t*able uses WebAssembly as the backend, compiling SQL code into WebAssembly and finally into machine code [18]. Given that the authors of [26] already implemented their algorithm into mu*t*able, we were able to reuse parts of their implementation, particularly the code generation. The implementation can be found in [3].

Analyzed Algorithms. In our benchmarks, we consider $\text{ResultDB}_{\text{Decompose}}$ (which uses DP_{CCP} for the whole enumeration) and the native $\text{ResultDB}_{\text{Semi-Join}}$ as baseline [26]. For acyclic queries, we compare the baseline with TD_{Root}. For cyclic queries, we also compare the baseline with TD_{Root}, but utilize either the folding of TD_{Fold}, or $\text{TD}_{\text{Fold-NoTVC}}$, where the latter is a reduced version of TD_{Fold} which does not utilize any two-vertex cuts in its enumeration. Notice that for TD_{Fold}, we utilize a simple $O(|V||E|)$ implementation to identify two-vertex cuts, which yields a better performance for the smaller node counts we use in the experiments. Further, we also analyze the folds generated by state-of-the-art GHD heuristics [1, 32]. Both share the fact that they first enumerate all possible GHDs, and choose a list of candidates which are within the range of the best *fractional hypertree width* [1], which is based on the AGM bound [2]. Afterward, in [1], heuristics are utilzed to decide on the final GHD, whereas in [32], a cost model is utilized. We label the heuristic-based algorithm as $\text{GHD}_{\text{Heuristic}}$ and the cost model-based algorithm, which additionally estimates the join and semi-join costs, as $\text{GHD}_{\text{CostModel}}$. We optimized both GHD enumerations to ignore plans with Cartesian Products to increase their performance. Whenever we refer to TD_{Fold}, $\text{TD}_{\text{Fold-NoTVC}}$, $\text{GHD}_{\text{Heuristic}}$, or $\text{GHD}_{\text{CostModel}}$, we consider them in their combination with TD_{Root}. Further, for all queries, the baseline is compared with $\text{TD}_{\text{ResultDB}}$.

Datasets & Workloads. To evaluate the overall query execution times, we utilize three datasets. For all these workloads, we make use of pre-generated cardinality estimations to prevent suboptimal plans because of inaccurate estimations.

Synthetic Schema. This dataset consists of a table F with attributes f, d_1, and d_2, two tables D_i, $i \in \{1, 2\}$, with attributes f and a, and five additional tables A_i, $i \in \{1, 5\}$, with attributes a_1, a_2, and a_3. F contains the tuples $\{(j, i, i) | 0 \leq i, j \leq 1999\}$, D_1 and D_2 contain the tuples $\{(j, j \bmod 10) | j \leq 1999\}$ and the remaining tables contain the tuples $\{(i, i, i) | 0 \leq i \leq 9\}$. All previously mentioned attributes are 4 byte integers. Each table also has an additional attribute w, which is a 100 byte char. This dataset is then utilized in a so-called *TVC query*. A TVC query supports the application of TVC cuts and is defined as follows:

Definition 8.1 (TVC Query). A query $G = (V, E)$ is called a TVC query, when $V = \{0, \ldots, |V| - 1\}$, $|V| > 4$, and $E = E_1 \cup E_2$, where

$$E_1 = \left\{\{i, i+1\} | 0 \leq i < |V| - 1\right\} \cup \{|V| - 1, 0\}, \text{and}$$
$$E_2 = \left\{\{i, |V| - 1 - i\} | 1 \leq i \leq \left\lceil \frac{|V|}{2} \right\rceil - 2\right\}$$

The exact query is shown in Figure 7. In the query, we project upon all attributes.

JOB. We evaluated a subset of queries from the Join Order Benchmark (JOB) [22] benchmark, based on the IMDb dataset. We chose the same queries presented in [26]. Like in [26], we limited all attribute sizes to 100 bytes. This is because mutable only supports fixed attribute sizes, and greater sizes would quickly exhaust the linear memory allocator deployed by mutable due to WebAssembly's limitation to 16 GiB of memory. Additionally, many queries in JOB extensively utilize `LIKE`-operations, which are evaluated in quadratic time only in mutable. As this can potentially greatly alter the relative performances of the different algorithms, we decided to utilize pre-filtered data. Lastly, we removed aggregations, since DRQs currently only support simple SPJ queries. We primarily utilize JOB to evaluate the performances of TD_{Fold} and TD_{Root} compared to $\text{ResultDB}_{\text{Semi-Join}}$. For that, we utilize two benchmarks, one where all queries (which are actually α−acyclic) are treated as cyclic queries, and one where GYO reductions [4, 34] were used to transform each JOB query into an equivalent acyclic query.

CE. We also evaluated a subset of queries from the CE benchmark [8]. Originally, the CE benchmark was designed to test the performance of query optimizers, particularly the cardinality estimators, of graph databases and consists of both acyclic and cyclic queries [7]. The authors of [7] then translated the CE benchmark into SQL. From these translated queries, we utilize all queries from the query template **dblp_cyclic_q8**. In each query, we project upon all attributes. To mitigate the memory limitations of mutable, we reduced the size of all input tables by 50%, while ensuring a minimum amount of 100,000 tuples per table.

8.2 Enumeration Time

This subsection presents our enumeration time benchmarks, presented in Figure 8, where we report the median runtime of 20 executions (using a log scale) for four important query shapes while varying the number of vertices in the graph.

Acyclic Queries. The results for both acyclic graphs are very similar. $\text{ResultDB}_{\text{Semi-Join}}$ delivers the overall fastest enumeration time, closely followed by TD_{Root}, which has a slight overhead of up to 50% only. On the other hand, since $\text{ResultDB}_{\text{Decompose}}$ utilizes DP_{CCP} as enumeration algorithm, it is much slower than TD_{Root} and $\text{ResultDB}_{\text{Semi-Join}}$, especially for star queries. However, this also means that the combined scheme $\text{TD}_{\text{ResultDB}}$ has no meaningful runtime overhead to $\text{ResultDB}_{\text{Decompose}}$.

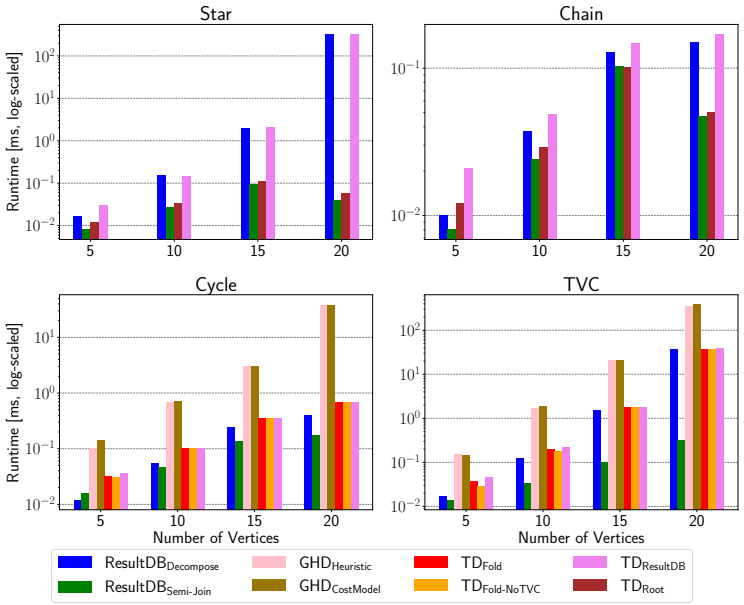


Fig. 8. Enumeration times for different query shapes.

Cyclic Queries. As you can see, $\text{ResultDB}_{\text{Semi-Join}}$ offers the fastest enumeration by creating two separate folds using a heuristic, enumerated independently using DP_{CCP}. Conversely, our proposed algorithms and $\text{ResultDB}_{\text{Decompose}}$ enumerate the entire graph, leading to an increased enumeration time compared to $\text{ResultDB}_{\text{Semi-Join}}$. Also, we can see that TD_{Fold} introduces a noticeable overhead compared to $\text{ResultDB}_{\text{Decompose}}$ in Cycle and TVC queries, caused by computing $C_{\text{Decompose}}$ and Algorithm 3. However, the relative overhead is significantly reduced by increasing the number of vertices. Further, we can see that including the greedy TVC folds in the TVC query does not meaningfully change the runtime of TD_{Fold} compared to $\text{TD}_{\text{Fold-NoTVC}}$, given that only one greedily folded graph will be created for each graph size. Additionally, one can see that GHD-based algorithms require much more enumeration time than the remaining algorithms. Overall, we can deduce that also in cyclic graphs, $\text{TD}_{\text{ResultDB}}$ introduces only little overhead compared to $\text{ResultDB}_{\text{Decompose}}$, especially for larger graphs.

Concluding, regarding **Q1**, our enumeration schemes only add small overheads to the existing baselines. For acyclic queries, TD_{Root} can almost match the runtime of $\text{ResultDB}_{\text{Semi-Join}}$. For cyclic queries, TD_{Fold} only introduces a visible overhead compared to $\text{ResultDB}_{\text{Decompose}}$ for small graphs, whereas in larger graphs the overhead is negligible. Lastly, $\text{TD}_{\text{ResultDB}}$ can unify all enumeration algorithms at very little additional cost, and beats GHD-based algorithms by a factor of up to 55x.

8.3 Query Execution Time

In this subsection, we present query execution time results for the queries from the synthetic, JOB, and CE datasets. In all experiments, we report the median execution time of five runs.

Synthetic Dataset. The results for varying selectivities of the attribute $F.f$ are presented in Figure 9. The graph shows the importance of enumeration to decide on the best folds. Joining the table F with any other relation causes significant redundancies, which severely hurts the performances of $\text{ResultDB}_{\text{Semi-Join}}$ and $\text{ResultDB}_{\text{Decompose}}$. Both TD_{Fold} and $\text{TD}_{\text{Fold-NoTVC}}$ manage to find better folds, namely the ones where F is not joined at all. We can also see that the plans generated

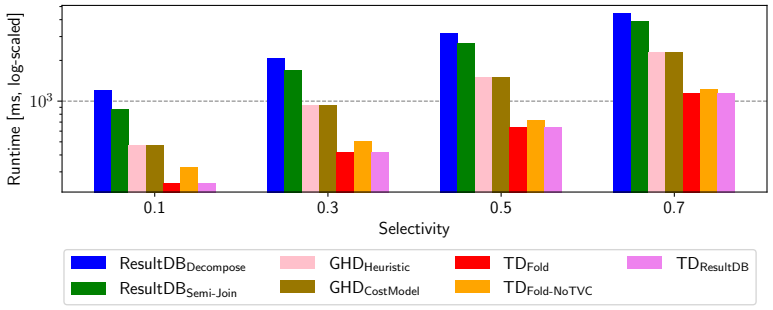


Fig. 9. Query execution times for the TVC query.

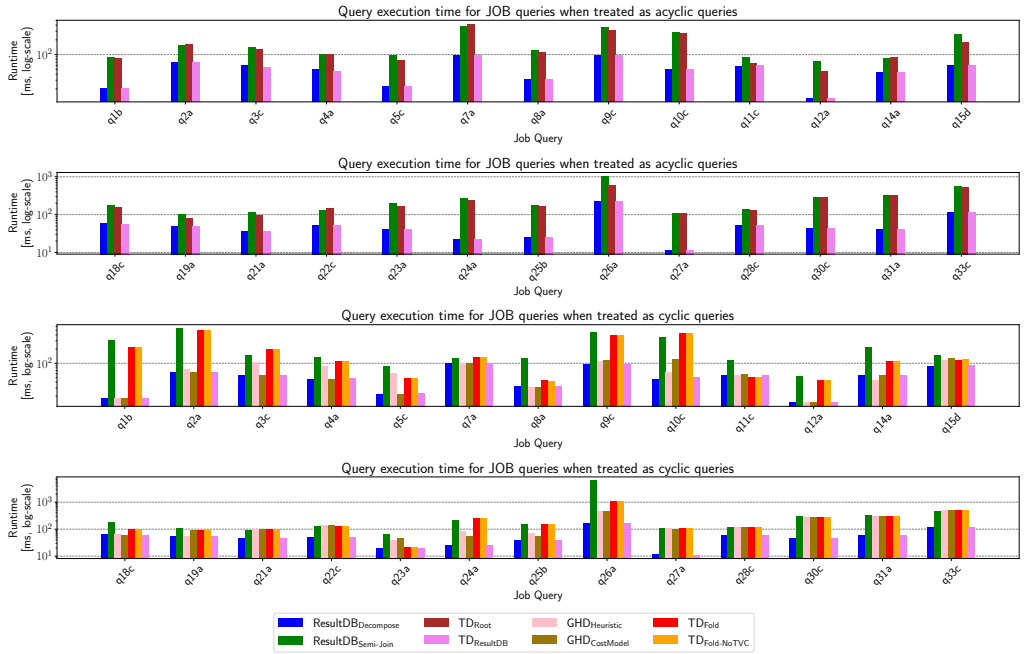


Fig. 10. Query execution time for JOB queries.

by TD$_{\text{Fold-NoTVC}}$ are slightly worse than those generated by TD$_{\text{Fold}}$, showing the importance of considering TVC folds during enumeration. Additionally, plans generated by our optimized algorithms have a faster performance than both GHD-based algorithms, demonstrating the significance of our new cost model C$_{\text{Fold}}$. Further, TD$_{\text{ResultDB}}$ always manages to find the best plans in each scenario.

JOB. We start by analyzing the results of acyclic JOB queries as shown in the two upper graphs of Figure 10. As visible, TD$_{\text{Root}}$ can find better plans than ResultDB$_{\text{Semi-Join}}$ for almost every query. The highest speed-up factor is at 1.7x in q26a, whereas we achieve an average speedup of 1.14x. Due to the small output sizes, ResultDB$_{\text{Decompose}}$ consistently produces the best plan in every query, which TD$_{\text{ResultDB}}$ successfully identifies. The results for cyclic queries are shown in the two lower graphs of Figure 10. Again, TD$_{\text{Fold}}$ is able to outperform ResultDB$_{\text{Semi-Join}}$ for almost all queries, up to a factor of 6x in q26a, and averaging at about 1.5x. Since no TVCs are in any JOB query, TD$_{\text{Fold}}$ cannot perform any better than TD$_{\text{Fold-NoTVC}}$. We can also see that GHD-based algorithms perform better than TD$_{\text{Fold}}$ for most queries as they consider joins with relations outside of cycles

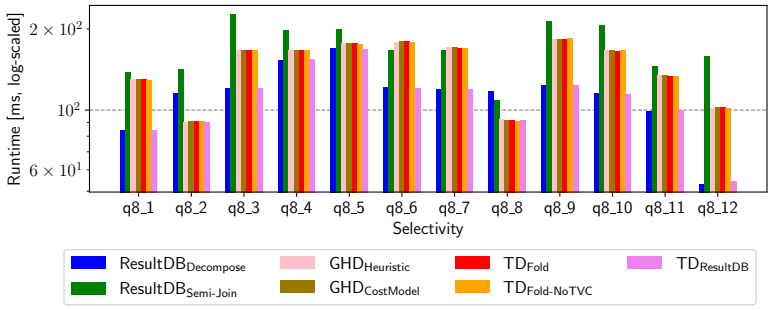


Fig. 11. Query execution time for the CE template dblp_cyclic_q8.

and TD$_{\text{Fold}}$ is aimed at redundancy-heavy queries, which JOB is not. Further, plans generated by GHD$_{\text{CostModel}}$ are on average better than those generated by GHD$_{\text{Heuristic}}$, aligning with results from [32]. Still, ResultDB$_{\text{Decompose}}$ always produces the best plans, whereas TD$_{\text{ResultDB}}$ can detect this.

CE. When looking at the CE benchmark in Figure 11, we can see that there are queries (q8_2 and q8_8) where TD$_{\text{Fold}}$ performs better than ResultDB$_{\text{Decompose}}$. This is an important result, as it underlines the importance of our new enumeration approach. Further, for q8_2, this was only made possible via optimizations, as ResultDB$_{\text{Semi-Join}}$ performs worse than ResultDB$_{\text{Decompose}}$ for this query. Further, plans generated by GHD$_{\text{Heuristic}}$ and GHD$_{\text{CostModel}}$ have the same quality than the plans generated by TD$_{\text{Fold}}$, demonstrating that even though TD$_{\text{Fold}}$ considers less plans, the heuristic chosen to prune this search space, namingly only considering folds within cycles, is effective for cyclic queries with redundancies. Other than that, the CE benchmark shows similar results to JOB. TD$_{\text{Fold}}$ can almost always produce a better plan than ResultDB$_{\text{Semi-Join}}$, up to a factor of 1.6x in q8_12, and averaging at 1.2x. Again, no TVCs were present, and by that, TD$_{\text{Fold}}$ and TD$_{\text{Fold-NoTVC}}$ perform equally well. For all queries (except q8_4), TD$_{\text{ResultDB}}$ can detect the best plans.

Conclusively, regarding **Q2**, one can say that our algorithms can increase the generated plan quality significantly compared to ResultDB$_{\text{Semi-Join}}$. However, for queries with low join cardinalities, ResultDB$_{\text{Decompose}}$ always generates the best plans. That being said, our new cost model is strong enough such that TD$_{\text{ResultDB}}$ can always determine the best presented plans, and always beat or match the plans generated by state-of-the-art GHD heuristics.

9 Conclusion and Future Work

The current state-of-the-art [26] to compute DRQs consists of two isolated algorithms, ResultDB$_{\text{Semi-Join}}$ and ResultDB$_{\text{Decompose}}$. We propose the new enumeration algorithms TD$_{\text{Root}}$ and TD$_{\text{Fold}}$ to greatly enhance plans for ResultDB$_{\text{Semi-Join}}$. These are unified together with ResultDB$_{\text{Decompose}}$ into TD$_{\text{ResultDB}}$, allowing to decide between plans generated by TD$_{\text{Root}}$, TD$_{\text{Fold}}$, and ResultDB$_{\text{Decompose}}$ for a given query.

Our experiments demonstrate the efficiency of our proposed algorithm(s). The enumeration time required by TD$_{\text{ResultDB}}$ introduces only a small overhead over the state-of-the-art for DRQs. More importantly, the plans generated by TD$_{\text{ResultDB}}$ vastly outperform those generated by ResultDB$_{\text{Semi-Join}}$ by up to 6x. Further, the cost models used for TD$_{\text{ResultDB}}$ are precise enough to decide between the different generated plans, and can even beat state-of-the-art GHD heuristics while offering smaller enumeration times.

However, there is still much work ahead. It is a very interesting direction for future work to study the intersection of the novel DRQ problem and the classical full reducer problem, and how both problems could benefit from each other. For example, one could analyze the enumeration

approach from TD_{Fold} in the context of full reducers. Furthermore, one should investigate how an increased search space for cycle resolutions (as present in, e.g., GHDs [1, 32]) might help in our approach. For example, different TVC choices in TD_{Fold}, as well as joins outside of cycles could be considered. To mitigate the effects of these increased search spaces, one could also analyze pruning opportunities for $TD_{ResultDB}$. In particular, branch-and-bound pruning [10] can be used to reduce the number of enumerated trees, join orders, and cycle solutions, especially when evaluating new subblocks during the folding of TVCs. For example, when all solutions for a BEP of a subblock are more expensive than the best plan for the original block, then all BEP sets containing said BEP can be ignored.

Further, one could investigate optimizations for DRQs depending on their use case. For example, an interesting optimization problem involves deciding, in distributed DBMSs, whether to send the query result as a single table, or as a database. The solution to that problem would be non-trivial given the high number of influencing factors, e.g., the achieved compression ratios, the overhead caused by the post-join and the DRQ computation itself, network conditions, and the available client system(s). This might be especially relevant when considering a batch of different queries, whose results could be compressed into a single database, while utilizing classical multi-query optimization techniques to speed up the DRQ computation. Another interesting use case would be to investigate optimization potential in the context of data provenance, where both the single-table result as well as a DRQ have to be computed. Again, one could analyze the benefits of having multiple queries share the same result database.

Finally, one could extend the scope of our optimizations and DRQs in general to also deal with more complex queries, e.g., semi-joins, outer-joins and data transformations, building upon the ideas already presented by Nix and Dittrich [26]. While semi-joins are straightforward to realize in the context of DRQs, outer-joins are more difficult as you also have to consider the order in which reductions are to be applied to prevent changing the result database. Data transformations would also require heavy modifications, as Nix and Dittrich [26] envisioned arbitrary transformations, meaning the creation of completely new tables alongside aggregations, opening up a new optimization area, as folds would now be required for acyclic queries too.

Acknowledgments

We thank Luca Gretscher, Marcel Maltry, and Joris Nix for their feedback and help preparing this paper. We also want to thank the anonymous reviewers for their valuable and constructive feedback. ChatGPT was utilized to help with spelling, grammar, and overall refinement of the writing of this work.

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 431–446. https://doi.org/10.1145/2882903.2915213

[2] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440

[3] Anonymous Author(s). 2025. Query Optimization for Database-Returning Queries. (2025). https://anonymous.4open.science/r/mutable-QO-ResultDB-D324

[4] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the Desirability of Acyclic Database Schemes. *J. ACM* 30, 3 (1983), 479–513. https://doi.org/10.1145/2402.322389

[5] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40. https://doi.org/10.1145/322234.322238

[6] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. 1981. Query Processing in a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.* 6, 4 (1981), 602–625. https://doi.org/10.1145/319628.319650

[7] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (2024), 3215–3228. https://www.vldb.org/pvldb/vol17/p3215-birler.pdf

[8] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Kenneth Salem. 2022. Accurate Summary-based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1533–1545. https://doi.org/10.14778/3529337.3529339

[9] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings (Lecture Notes in Computer Science)*, Georg Gottlob and Moshe Y. Vardi (Eds.), Vol. 893. Springer, 54–67. https://doi.org/10.1007/3-540-58907-4_6

[10] David DeHaan and Frank Wm. Tompa. 2007. Optimal top-down join enumeration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 785–796. https://doi.org/10.1145/1247480.1247567

[11] Ronald Fagin. 1983. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM* 30, 3 (1983), 514–550. https://doi.org/10.1145/2402.322390

[12] Pit Fender and Guido Moerkotte. 2011. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 864–875. https://doi.org/10.1109/ICDE.2011.5767901

[13] Jörg Flum, Markus Frick, and Martin Grohe. 2002. Query evaluation via tree-decompositions. *J. ACM* 49, 6, 716–752. https://doi.org/10.1145/602220.602222

[14] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 57–74. https://doi.org/10.1145/2902251.2902309

[15] Carsten Gutwenger and Petra Mutzel. 2000. A Linear Time Implementation of SPQR-Trees. In *Graph Drawing, 8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, September 20-23, 2000, Proceedings (Lecture Notes in Computer Science)*, Joe Marks (Ed.), Vol. 1984. Springer, 77–90. https://doi.org/10.1007/3-540-44541-2_8

[16] Immanuel Haffner and Jens Dittrich. 2023. Efficiently Computing Join Orders with Heuristic Search. *Proc. ACM Manag. Data* 1, 1 (2023), 73:1–73:26. https://doi.org/10.1145/3588927

[17] Immanuel Haffner and Jens Dittrich. 2023. mutable: A Modern DBMS for Research and Fast Prototyping. 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/p41-haffner.pdf

[18] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. 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, 1–13. https://doi.org/10.48786/EDBT.2023.01

[19] Frank Harary. 1969. *Graph Theory*. Addison-Wesley Publishing Company, New York.

[20] John E. Hopcroft and Robert Endre Tarjan. 1973. Dividing a Graph into Triconnected Components. *SIAM J. Comput.* 2, 3 (1973), 135–158. https://doi.org/10.1137/0202012

[21] John E. Hopcroft and Robert Endre Tarjan. 1973. Efficient Algorithms for Graph Manipulation [H] (Algorithm 447). *Commun. ACM* 16, 6 (1973), 372–378. https://doi.org/10.1145/362248.362272

[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[23] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 122–135. https://doi.org/10.1145/3514221.3517871

[24] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. (2006), 930–941. http://dl.acm.org/citation.cfm?id=1164207

[25] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini (Eds.). ACM, 37–48. https://doi.org/10.1145/2213556.2213565

[26] Joris Nix and Jens Dittrich. 2025. Extending SQL to Return a Subdatabase. In *ACM SIGMOD International Conference on Management of Data (SIGMOD), June 2025.* https://bigdata.uni-saarland.de/publications/Nix,%20Dittrich%20-%20Extending%20SQL%20to%20Return%20a%20Subdatabase.pdf

[27] Dan Olteanu and Jakub Zavodny. 2012. Factorised representations of query results: size bounds and readability. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, Alin Deutsch (Ed.). ACM, 285–298. https://doi.org/10.1145/2274576.2274607

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

[29] Jens M. Schmidt. 2013. A simple test on 2-vertex- and 2-edge-connectivity. *Inf. Process. Lett.* 113, 7 (2013), 241–244. https://doi.org/10.1016/J.IPL.2013.01.016

[30] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2, 146–160. https://doi.org/10.1137/0201010

[31] Robert Endre Tarjan. 1974. A Note on Finding the Bridges of a Graph. *Inf. Process. Lett.* 2, 6 (1974), 160–161. https://doi.org/10.1016/0020-0190(74)90003-9

[32] Qichen Wanga, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. In *ACM SIGMOD International Conference on Management of Data (SIGMOD), June 2025.* https://qichen-wang.github.io/publication/SIGMOD2025

[33] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

[34] C. T. Yu and M. Z. Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*. IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509

Received April 2025; revised July 2025; accepted August 2025