

Optimizing Linearized Join Enumeration by Adapting to the Query Structure

Altan Birler ¹, Mihail Stoian ², and Thomas Neumann ¹

Abstract:



In database systems, join enumeration is a critical but NP-hard problem, especially challenging for queries with a large number of joins, as often found in graph workloads. To manage this complexity, some state-of-the-art methods reduce the search space by employing query graph linearization. However, even after linearization, join enumeration still has cubic runtime, which can be inefficient for very large queries. We propose an improved enumeration algorithm that dynamically adapts to the query graph structure, avoiding the generation of invalid or redundant plans. This reduces the time complexity for star queries from $O(n^3)$ to $O(n \log n)$ and enables the optimization of general tree queries of thousands of joins within seconds.


Keywords: query optimization, join enumeration, dynamic programming, very large join queries

1 Introduction

In database systems, information is stored across multiple relations to avoid redundancy and ensure data integrity. Graph workloads take this to the extreme by normalizing the schema to the highest possible degree, by expressing each “attribute” as a separate “edge”. This results in a large number of relations that need to be joined back together to answer various queries. The order in which these joins are executed can have a significant impact on the query performance. However, finding the optimal join order is NP-hard [IK84], which makes it impractical to find optimal orders for queries with more than 20 relations in the general case, necessitating an effective fallback strategy.

In real-world scenarios, this challenge is evident in queries involving hundreds or even thousands of relations. For example, the recently published Redset dataset, which provides a sample of user queries from Amazon Redshift, includes a query that accesses 2,296 relations [Re24]. Similarly, operational BI reporting applications can have “mega queries” with possibly 1,000 relations in their FROM clause [Di09]. Additionally, in SAP HANA’s workloads, certain complex database views, after unfolding all referenced views, contain up to 4,598 relations, with 161 views referencing more than 100 relations each [MBL17].

¹ Technische Universität München, CIT, Boltzmanstr. 3, 85748 Garching bei München, Germany, altan.birler@tum.de,  <https://orcid.org/0009-0007-1177-772X>; neumann@in.tum.de,  <https://orcid.org/0000-0001-5787-142X>

² UTN, Department of CS & AI, Ulmenstr. 52i, 90443 Nuremberg, Germany, mihail.stoian@utn.org,  <https://orcid.org/0000-0002-8843-3374>

These examples underscore the necessity of finding efficient join orders in polynomial time, as optimal solutions are infeasible for such large queries.

Current state-of-the-art approaches like Linearized Dynamic Programming (LinDP) attempt to reduce this complexity by *linearizing* the query graph. After the linearization, commutativity of joins are disallowed, and only associativity is considered, i.e., the optimal parenthesization is constructed. The parenthesization step, as described by Neumann and Radke [NR18], takes $\Theta(n^3)$ time, regardless of the query’s structure, producing many parenthesizations with invalid joins. Most queries are not cliques, and the number of valid joins can often be significantly smaller. Queries with a star schema, for example, have only $O(n)$ valid join pairs after linearization. This implies that there is significant room for improvement in adapting the parenthesization step to the query graph structure.

Contribution. In this work, we propose a dynamic programming (DP) parenthesization algorithm that adapts itself to the query graph structure and generates exactly the set of all valid plans. Our approach enumerates valid joins pairs optimally up to a single log factor, reducing the time complexity from $O(n^3)$ to $O(n \log n)$ for star queries. We evaluate our algorithm on a set of synthetic query shapes and show that it outperforms state-of-the-art LinDP by orders of magnitude in terms of runtime. We additionally propose *linearization transfer*, a novel technique to further reduce redundant work by reusing shared parts of linearizations across parenthesizations. These two approaches combined are able to adaptively exploit the query structure and scale to finding optimal join orders for *thousands* of relations under a second for a large class of queries.

2 Related Work

The goal of join enumeration is to efficiently explore the problem’s search space while ensuring the optimal solution is found. This is particularly important for large join queries since ensuring optimality is known to be NP-hard [IK84]. This led to the development of several exact, exponential-time algorithms for small-size queries [Ch95; DT07; FM11; FM12; HD23; Ma22; MN06; Se79; SK24; TK17; VM96], and of heuristics-based algorithms for medium-size and large-size queries [Ch09; Di09; Fe98; KS00; NR18; Sw89]. A new line of research uses quantum solvers due to the increasing effectiveness of quantum annealers in recent years [Fr24; SSM23; STM23; Wi23]. In the following, we only detail on related work that inspired our new enumeration algorithm.

Exact Algorithms. We will start with the exact algorithms, since they went through a similar transformation as LinDP will undergo in this work. In a seminal paper, Selinger et al. [Se79] introduced the join enumeration problem and the first exact, exponential-time algorithm, DPsize. The main drawback: the algorithm ran in time $O(4^n)$. Vance and Maier [VM96] observed this limitation and proposed DPsub that runs in time $O(3^n)$ in the worst case. The inherent limitation of DPsub is that its runtime is exponential regardless of the query structure. To this end, Ono and Lohman [OL90] analyzed the minimum number of subplan

pairs, referred to as *connected complemented pairs* (ccp), that have to be iterated by any dynamic program. Motivated by this, Moerkotte and Neumann [MN06] introduced DPccp, which matches the time-bound proposed by the previous work. Their key insight is that one can modify DPsub so that it adapts to the query structure. Namely, DPccp will enumerate only those subplans that are actually needed, i.e., it avoids generating invalid plans. Our work takes its inspiration from this very fact.

Linearized Dynamic Programming. The NP-hardness of the problem has led researchers to consider non-exact algorithms. After a long line of research [BGJ10; Ch09; Di09; Fe98; SMK97; Sw89], the state of the art is to use *search space linearization* [NR18]. The key idea is to exploit two prominent algorithms in the literature: (a) The optimal algorithm for acyclic queries, restricted to the setting of left-deep join trees [IK84; KBZ86], and (b) the ad-hoc optimal algorithm for chain queries. Neumann and Radke [NR18] proposed LinDP as a two-stage algorithm: First run (a), which returns a chain-like arrangement of the relations—this is what linearization refers to—and then run (b), which builds the optimal bushy join tree on top of the given, *fixed* linearization.

The main advantage of this method is that if the linearization happens to be the order of the leaf nodes of an optimal join tree, then this algorithm is indeed optimal. The main disadvantage is that LinDP runs in cubic time *per* linearization, making it prohibitively expensive for large queries. Instead, the standard greedy algorithm, GOO [BGJ10; Fe98], optimizes arbitrarily large queries in $O(n^2 \log n)$ -time. To obtain better plans, one can use IDP [KS00], which refines the greedy plan by running the exact algorithm on subplans of size up to k . Naturally, this limits k to a handful of relations. Given the effectiveness of LinDP, Neumann and Radke [NR18] suggested replacing the exact algorithm in IDP by LinDP, thus being able to increase k to 100 and further refine the initial plan. Our work suggests that the greedy step is no longer necessary for a large class of queries.

3 Background

In this section, we introduce the necessary notation and concepts used throughout the paper. In Sec. 3.1, we define the join enumeration problem and then outline the IKKBZ algorithm in Sec. 3.2, which is used to compute the linearizations used in LinDP. We also provide an overview of the parenthesization problem in Sec. 3.3, which is utilized to build join trees from linearizations. Finally, we outline LinDP [NR18] in Sec. 3.4, which combines the linearization and parenthesization steps to find good join orders in polynomial time.

3.1 Join Enumeration

Basic Notation. In this work, we consider SPJ queries, where the only relational operators allowed are σ (selection), Π (projection), and \bowtie (join) [Co70] and ignore aggregations [FBN23].

Consequently, the predicates can either be join predicates, e.g., $R_1.a = R_2.b$, or selection predicates, e.g., $R_1.a = \text{const}$, where $\{R_i\}_{i \in [n]}$ is the set of the n relations considered in the query. To express how large a relation is, we employ the cardinality function $|\cdot|$. For instance, $|R_1| = 10^3$ specifies that R_1 has 10^3 tuples. We use the cardinality function also for joins or cross products, e.g., $|R_1 \bowtie R_2|$ represents the cardinality of the (natural) join $R_1 \bowtie R_2$.

Query Graph. A SQL query is modeled as a query graph $G = (V, E)$, where the relations are represented by vertices and the join predicates as the edges between these. The most common query graph classes are stars, chains, snowflakes, arbitrary trees, cycles, and cliques, each incurring different runtime complexities for the join enumeration problem [HD23; MN06; Ne09; NR18]. For instance, while chains can be optimized exactly in polynomial time, it is NP-hard to optimize cliques exactly [MS96].

We visualize a query graph corresponding to a SQL query in Fig. 1. The original query consists of four relations, $\{R_1, R_2, R_3, R_4\}$, and three join predicates, forming a chain query.

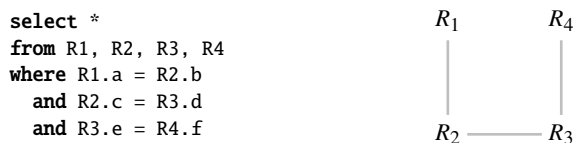


Fig. 1: A SQL query (left) and its corresponding query graph (right). The vertices represent the relations and the edges the corresponding joins between them.

Join Tree. The output of join enumeration can be concisely represented as a *join tree* (or, alternatively, a query plan). A join tree is a rooted binary tree where the leaf nodes are the relations, and the inner nodes are the join operators. The classes of join trees we will be dealing with are *left-deep* and *bushy*. Similar to the query graph shape, the join tree shape also has an influence on the time complexity of the problem, i.e., the optimal left-deep tree can be found in polynomial time using the IKKBZ algorithm.

3.2 Computing Linearizations via IKKBZ

The current state-of-the-art algorithm, LinDP, uses the notion of *linearizations*. We next describe how these are computed (later, in Sec. 4.6, our linearization transfer technique will exploit this very construction). Namely, the linearizations are given by the IKKBZ algorithm which guarantees optimal left-deep join trees for acyclic queries [IK84; KBZ86].

Precedence Graph. The key idea behind IKKBZ is that, indeed, an acyclic query induces a partial order on the relations, by rooting the graph in one of the vertices. This is what a *precedence graph* refers to. This perspective makes the ordering problem easier, since the algorithm can solve the (local) problem on all n possible precedence graphs. The global optimum is then the best of all local solutions.

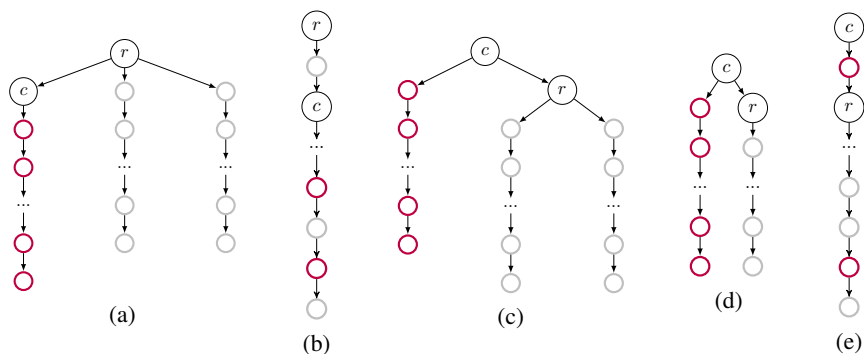


Fig. 2: (a) Precedence graph of r . (b) Linearization of the precedence graph of r . (c) The precedence graph of c , a child of r : We simply need to *rotate* the edge $\{r, c\}$, promoting c to the root. (d) The linearization of the subtree of r can be obtained from that in step (b) by *splitting* the nodes in the original subtree of c . (e) The linearization of the precedence graph of c is obtained by finally merging the two chains in the previous step.

Algorithm. Once the precedence graph has been obtained for a given relation R_i , i.e., the query graph has been rooted in R_i , we can *linearize* it and obtain the optimal left-deep solution for the respective relation. This is done in a bottom-up strategy, namely: Recursively linearize the subtrees and merge them based on a certain rank function. The original algorithm [IK84] has been subsequently improved in [KBZ86], by lowering the runtime from $O(n^2 \log n)$ to $O(n^2)$. In this work, we will use this version. For completeness, we outline its main steps in the following.

The main idea is that, instead of independently building the precedence graphs for all n relations, one can reuse solutions across them. Let r be the root of the current precedence graph and c one of its children, as shown in Fig. 2(a); its linearization is shown in Fig. 2(b). Then, the precedence graph of c , visualized in Fig. 2(c), can be constructed by *rotating* the edge $\{r, c\}$, promoting node c to be the root of the new precedence graph. This is done in two phases, outlined in Fig. 2(d) and Fig. 2(e), respectively: To linearize the subtree of r (the gray-colored chains), we reuse the old linearization in Fig. 2(b) by *splitting* the nodes in the original subtree of c (the red-colored nodes). Finally, we merge the two remaining chains according to the ranks and obtain the linearization of the precedence graph of c .

3.3 Parenthesization

In this subsection, we introduce the parenthesization problem, which is a key component of the LinDP algorithm. Given a fixed order of relations, such as the linearization produced by IKKBZ, finding an optimal join tree corresponds to the optimal parenthesization, i.e., the

optimal order in which to execute operations that are associative but not commutative. This problem can be expressed as the following cost minimization:

$$c(i, i) = 0, \tag{1}$$

$$c(i, j) = w(i, j) + \min_{i \leq k < j} \{c(i, k) + c(k + 1, j)\}, \tag{2}$$

where $c(i, j)$ is the cost of parenthesizing the query $R_i \bowtie \dots \bowtie R_j$, and $w(i, j)$ is the cardinality of the join $R_i \bowtie R_{i+1} \bowtie \dots \bowtie R_j$.

This optimization problem admits a dynamic programming solution, where the cost of parenthesizing the query $R_i \bowtie \dots \bowtie R_j$ is computed by selecting an optimal split point $k \in [i, j)$ corresponding to the query plan $(R_i \bowtie \dots \bowtie R_k) \bowtie (R_{k+1} \bowtie \dots \bowtie R_j)$, and computing the optimal plan for the corresponding subranges recursively. The pseudocode for this algorithm is shown in Fig. 3 with both top-down and bottom-up variants. While the top-down variant is a more direct translation of the recursive definition, the bottom-up variant is more efficient, as it avoids redundant computations and checks by always computing the intermediate results before they are needed. Unlike the original implementation of LinDP [NR18], the bottom-up variant we show iterates from right to left with i from $n - 1$ to 0. For every i it iterates over all adjacent valid range pairs $[i, k]$ and $[k + 1, j]$ with increasing k and increasing j . By the time we consider a new plan for range $[i, j]$ in Line 24, the optimal plans for the required subranges $[i, k]$ and $[k + 1, j]$ have already been computed.

The parenthesization problem is well studied, and allows for various optimization for different $w(i, j)$ functions, assuming $w(i, j)$ has certain exploitable properties and can be computed in constant time [Ya80]. In the context of matrix chain multiplication, the parenthesization problem is known to be solvable in $O(n \log n)$ time, where n is the number of matrices [HS82; HS84]. However, in the context of join enumeration, no improvement to the naive $O(n^3)$ enumeration algorithm is known. Additionally, computing $w(i, j)$ does not take constant time, as even checking connectivity of a set of relations is nontrivial. We will address these limitations in Sec. 4.

3.4 Linearized Dynamic Programming

We have discussed how IKKBZ [IK84; KBZ86] computes the optimal left-deep join trees for acyclic queries. We have also introduced the parenthesization problem, which is used to build bushy join trees for a given relation order. Dynamic programming can be used to compute the optimal parenthesization for a given order of relations in polynomial time, which for a chain query corresponds to the optimal bushy tree. The LinDP algorithm combines these two steps to find good join orders in polynomial time for a large class of queries, whereby the order of the relations produced by IKKBZ is used as the input linearization to the parenthesization step.

The IKKBZ algorithm produces one left-deep tree for each start relation. Among these trees, the one with the lowest cost is selected as the optimal left-deep tree. The LinDP algorithm

```

1 def topDown(i = 0, j = n - 1):
2     if i == j:
3         return 0
4     result = infity
5     for k in range(i, j):
6         if not linked(i, k, j):
7             continue
8         result = min(result, topDown(i, k) + topDown(k + 1, j) + w(i, j))
9     return result
10
11 def bottomUp():
12     c = [[infity for _ in range(n)] for _ in range(n)]
13     for i in range(n):
14         c[i][i] = 0
15     for i in range(n - 1, -1, -1):
16         for k in range(i + 1, n):
17             if c[i][k] == infity:
18                 continue
19             for j in range(k + 1, n):
20                 if c[k + 1][j] == infity:
21                     continue
22                 if not linked(i, k, j):
23                     continue
24                 c[i][j] = min(c[i][j], c[i][k] + c[k + 1][j] + w(i, j))
25     return c[0][n - 1]

```

Fig. 3: Top-down recursive and bottom-up dynamic programming solution for the parenthesization problem. The `linked` function checks whether the ranges $[i, k]$ and $[k + 1, j]$ are connected over an edge in between those ranges. The `bottomUp` approach is more efficient, as it executes the operations in an order where required intermediate results are always available when they are needed. This efficiently avoids redundant computations. Note that, if naively implemented, each individual call to `linked` can take $O(n^2)$ time raising the total cost of computation to $O(n^5)$.

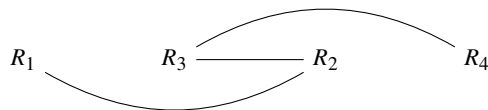


Fig. 4: Example of a chain query graph where the linearization is not a linear chain.

as described by Neumann and Radke [NR18] then computes the optimal parenthesization for the linearization of the optimal left-deep tree. However, this resulting algorithm may produce suboptimal plans for chain queries, as the optimal linearization of a chain query may not be a linear chain as shown in Fig. 4.

To fix this issue, the existing implementation of `LinDP` in the Umbra database system [NF20] computes the optimal parenthesization for *each* linearization produced by `IKKBZ`. The algorithm then selects the join tree with the lowest cost among all the computed parenthesizations. We will call this variant of the algorithm `extended LinDP`. The resulting algorithm produces optimal join trees for star and chain queries, and is a good heuristic for general tree or cyclic queries. The relationships between different graph types are visualized in Fig. 5.

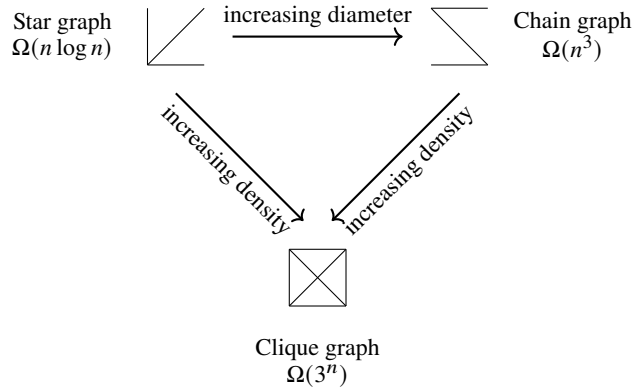


Fig. 5: Diagram of graph types and their respective time complexities of best-known algorithms for join enumeration. The star graph has a time complexity of $n \log n$ using `IKKBZ`, the chain graph has a time complexity of n^3 using parenthesization, and the clique graph has a time complexity of 3^n using dynamic programming. `Extended LinDP` produces optimal plans for stars and chains, but not necessarily for arbitrary trees in between.

4 Approach

In this section, we describe our *adaptive LinDP* algorithm, with two key optimizations over *extended LinDP*:

1. How to modify parenthesization to avoid generating invalid plans.
2. How to use shared suffixes in linearizations to avoid redundant computations.

In Sec. 4.1 we will discuss invalid plans and why they exist. Then, in Sec. 4.2, we will prove Lemma 1 which leads to a more efficient computation of `linked(i, k, j)` for the baseline DP algorithm in Fig. 3. We will further build upon that in Sec. 4.3 and 4.4, by describing our efficient algorithm which avoids iterating over invalid plans, by using a data structure we describe in Sec. 4.5. Finally, in Sec. 4.6, we describe linearization transfer, our second key

optimization. The resulting algorithm is a strict improvement over LinDP. We find the same query plan as LinDP, but often orders of magnitude faster.

4.1 Valid and Invalid Parenthesizations

Among parenthesization problems, what is unique about the join ordering setting is that many ranges $[i, j]$ are invalid, i.e., they have $w(i, j) = +\infty$. Consider the join query given in Fig. 6. If we are to only allow associativity and not commutativity, the parenthesization $R1 \bowtie (R2 \bowtie R3)$ is invalid, as it would require a join predicate between $R2$ and $R3$. Since there is no such predicate, only the parenthesization $(R1 \bowtie R2) \bowtie R3$ is valid.³ Taken to the extreme, a star query graph of n relations has $\Theta(n)$ valid parenthesizations,⁴ while the best-known algorithm⁵ for determining the optimal parenthesization has runtime $\Theta(n^3)$.

```

select *
from R1, R2, R3
where R1.a = R2.a and R1.b = R3.b

```

Fig. 6: Example of a star query graph with invalid parenthesizations. Relations R_2 and R_3 are neighboring, but are not connected over a predicate.

To this end, we define a range of relations $[i, j]$ corresponding to the relations $R_i \bowtie R_{i+1} \bowtie \dots \bowtie R_j$ to be valid if it can be decomposed into two adjacent subranges that are themselves valid and can be joined together using an existing predicate. More formally:

$$\begin{aligned}
 \text{linked}(i, k, j) &= \exists a \in [i, k], b \in [k + 1, j]: \text{edge}(a, b), & \forall i \leq k < j, \\
 \text{valid}(i, i) &= \text{true}, & \forall i, \\
 \text{valid}(i, j) &= \exists k \in [i, j]: \text{valid}(i, k) \wedge \text{valid}(k + 1, j) \wedge \text{linked}(i, k, j), & \forall i < j,
 \end{aligned}$$

where $\text{edge}(a, b)$ indicates whether there is a predicate between relations R_a and R_b . Note that a range may be connected but invalid as in Fig. 7. We do not consider such ranges, as only those ranges that can be recursively decomposed into valid ranges can be used to build a valid join tree. Additionally, even though we assume that the full query itself is connected, i.e., there is a path between any two relations following the join predicates, the graph corresponding to a subrange may be arbitrarily complicated. This makes the problem of finding valid parenthesizations non-trivial. To demonstrate that the graph can be made arbitrarily complicated, we can simply take an arbitrary query, not necessarily connected,

³ R_2 and R_3 may be joined with a cross product. However, cross products are typically disabled in join optimizers as they are rarely beneficial and often detrimental. Enabling them can mislead the optimizer due to cardinality underestimations, causing costly mistakes [RN19]

⁴ Here, we assume that the *hub* relation, the relation connected to all other relations, is one of the first two relations in the linearization. IKKBZ is guaranteed to produce such a linearization, as otherwise, the corresponding left-deep tree would be invalid.

⁵ Here, we refer to the algorithm due to Neumann and Radke [NR18]. Note that our implementation of the baseline DP algorithm in Fig. 3 would have time complexity $O(n^2)$ as it eagerly checks if subranges are valid.

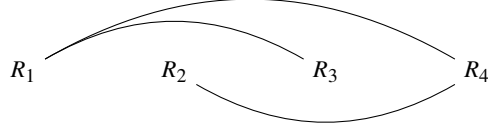


Fig. 7: Example of a connected but invalid range $[1, 4]$. There is no split point k such that both $[1, k]$ and $[k + 1, 4]$ are valid.

add a relation to the end, and connect it to every previous relation. This is demonstrated in Fig. 8, where the range $[1, 4]$ corresponds to an arbitrary graph.

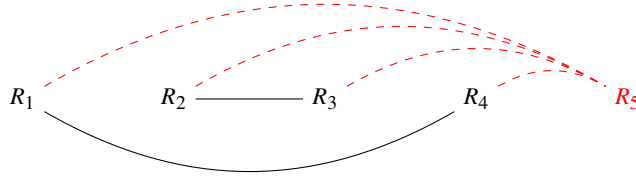


Fig. 8: Example of an arbitrary graph corresponding to the range $[1, 4]$. The entire query is made connected by the addition of R_5 .

Our approach will be to dynamically adapt the parenthesization to the query structure by only considering valid parenthesizations. This gives us a runtime of $O((m + c) \log n + p)$, where c is the number of valid ranges of relations, and p is the number of valid consecutive pairs of ranges. While the worst-case runtime will still be $O(n^3)$, e.g., when p itself is $\Omega(n^3)$, many classes of queries, such as stars and trees, will benefit from this optimization.

4.2 Efficient Computation of Linked

Now that we have introduced the fundamentals of join parenthesization, we will discuss how to efficiently compute the $\text{linked}(i, k, j)$ function used in Fig. 3, which computes whether the ranges $[i, k]$ and $[k + 1, j]$ are linked over an edge. We will compute this function once per an (i, j) pair with the first valid k , and reuse this result for other k values. This will give us an efficient baseline dynamic programming algorithm, which we will further improve in Sec. 4.3.

Lemma 1 For $i \leq j$, $\text{valid}(i, j) \implies \forall k \in [i, j]: \text{linked}(i, k, j)$.

Proof. We will prove by induction on the size $n = j - i$ of the range $[i, j]$ that if $\text{valid}(i, j)$, then $\forall k \in [i, j]: \text{linked}(i, k, j)$.

Base Case: For the base case $n = 0$, $\text{valid}(i, j) = \text{valid}(i, i)$ holds by definition.

Inductive Step: Assume that for all ranges of size $n' < n$, if $\text{valid}(i', j')$ for any range $[i', j']$, then $\forall k' \in [i', j']: \text{linked}(i', k', j')$.

Now, consider a range $[i, j]$ where $n = j - i$ such that $\text{valid}(i, j)$. By the definition of $\text{valid}(i, j)$, there exists a split point $k \in [i, j]$ such that:

$$\text{valid}(i, k) \wedge \text{valid}(k + 1, j) \wedge \text{linked}(i, k, j)$$

By the inductive hypothesis, since $k - i < n$ and $j - (k + 1) < n$, the ranges $[i, k]$ and $[k + 1, j]$ satisfy:

- $\forall k' \in [i, k]: \text{linked}(i, k', k) \implies \text{linked}(i, k', j)$
- $\forall k' \in [k + 1, j]: \text{linked}(k + 1, k', j) \implies \text{linked}(i, k', j)$

Since $\text{linked}(i, k, j)$ also holds, $\forall k' \in [i, j]: \text{linked}(i, k', j)$. □

Using Lemma 1, we can reduce redundant computations of $\text{linked}(i, k, j)$. For a given (i, j) , for the first k we find that satisfies $\text{valid}(i, k) \wedge \text{valid}(k + 1, j)$, we can compute $\text{linked}(i, k, j)$ in $\mathcal{O}(m)$ time by naively iterating over edges in that given range, where m is the number of edges. If $\text{linked}(i, k, j)$ holds, this implies $\text{valid}(i, j)$ and thus $\text{linked}(i, k', j)$ will hold for all $k' \in [i, j]$, meaning that we do not need to recompute $\text{linked}(i, k', j)$ for any other k' . If $\text{linked}(i, k, j)$ does not hold, this implies that $\text{valid}(i, j) = \text{false}$, implying that we can skip any further computation for this (i, j) pair. The optimized bottom-up enumeration algorithm is listed in Fig. 9.

Note that one could further optimize the computation of $\text{linked}(i, k, j)$ by utilizing binary search over sorted edge lists, which would allow for a $\mathcal{O}(n \log n)$ time complexity per $\text{linked}(i, k, j)$ invocation. This would reduce the time complexity from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3 \log n)$ for clique queries. However, this would not improve the overall time complexity for acyclic query graphs, as the time complexity would remain $\mathcal{O}(n^3)$.

4.3 Avoiding Invalid Plans

The LinDP algorithm iterates over a large number of potentially invalid ranges with disconnected relations. We propose an algorithm that adapts itself to the query graph structure and avoids generating invalid plans. This reduces the iteration time complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n \log n)$ in star query graphs. Additionally, our approach can explicitly produce a join edge connecting pairs of ranges at zero cost, which is particularly beneficial for cardinality estimation computations that would otherwise increase the runtime.

We will first start with how we can iterate over all valid ranges efficiently, then we will extend this approach to finding pairs adjacent valid ranges so that we can compute the optimal plan using dynamic programming. Given a valid range $[i, j]$ starting at i , we want to find the smallest $j' > j$ such that $[i, j']$ is a valid range. If we can always find the next

```

1 def bottomUp():
2     # validity
3     v = [[0 for _ in range(n)] for _ in range(n)]
4     # cost
5     c = [[infy for _ in range(n)] for _ in range(n)]
6     for i in range(n):
7         c[i][i] = 0
8     for i in range(n - 1, -1, -1):
9         for k in range(i + 1, n):
10            if c[i][k] == infy:
11                continue
12            for j in range(k + 1, n):
13                if c[k + 1][j] == infy:
14                    continue
15                # 0 = not computed, 1 = connected, 2 = not connected
16                if v[i][j] == 0:
17                    if linked(i, k, j):
18                        v[i][j] = 1
19                else:
20                    v[i][j] = 2
21                if v[i][j] == 1:
22                    c[i][j] = min(c[i][j], c[i][k] + c[k + 1][j] + w(i, j))
23    return c[0][n - 1]

```

Fig. 9: Bottom up dynamic programming solution for the parenthesization problem with efficient computation of the $\text{linked}(i, k, j)$ function. The linked function checks whether the ranges $[i, k]$ and $[k + 1, j]$ are linked over an edge in between those ranges.

```

1 def produceValidRanges():
2     for i in range(n - 1, -1, -1):
3         j = i
4         while j < n:
5             yield (i, j)
6             j = nextValidRange(i, j)

```

Fig. 10: We produce all valid ranges incrementally by finding the next valid range after j . We make sure to produce ranges with decreasing starting point and increasing ending point such that dynamic programming can properly utilize the results of previous iterations.

valid range efficiently, then we can iterate over all valid ranges incrementally. This concept is illustrated in Fig. 10.

We will now prove lemmas that will help us find the next valid range efficiently.

Lemma 2 Given $a \leq b \leq c \leq d$,

$$\text{valid}(a, c) \wedge \text{valid}(b, d) \implies \text{valid}(a, d)$$

Proof. We will prove the lemma by induction on the sum of the sizes of the two sub ranges $(c - a) + (d - b)$.

Base case: Given, $c - a = 0 \vee d - b = 0$, $\text{valid}(a, d)$ is trivially true.

Inductive step: Assume the lemma is true for all ranges with $c - a + d - b < n$. We will prove it for $c - a + d - b = n$.

We arbitrarily pick one of $[a, c]$ and $[b, d]$. Without loss of generality, we pick $[a, c]$. We can pick a split point $k \in [a, c]$, as $c - a > 0$, such that $\text{valid}(a, k) \wedge \text{valid}(k+1, c) \wedge \text{linked}(a, k, c)$. There are two possibilities. If $k < b$, given our induction hypothesis, $\text{valid}(k+1, c) \wedge \text{valid}(b, d)$ implies $\text{valid}(k+1, d)$. As $\text{valid}(a, k)$ and $\text{linked}(a, k, c)$ are known, $\text{valid}(a, d)$ is implied. If $k \geq b$, given our induction hypothesis, $\text{valid}(a, k) \wedge \text{valid}(b, d)$ directly implies $\text{valid}(a, d)$. \square

Lemma 3 Given $i \leq j$, $\text{valid}(i, j)$, and $j' > j$,

$$\text{linked}(i, j, j') \wedge (\exists k \in [i+1, j+1]: \text{valid}(k, j')) \implies \text{valid}(i, j')$$

Proof. There are two possibilities.

If $k \leq j$, according to Lemma 2:

$$\text{valid}(i, j) \wedge \text{valid}(k, j') \implies \text{valid}(i, j').$$

If $k = j + 1$:

$$\text{valid}(i, j) \wedge \text{valid}(j+1, j') \wedge \text{linked}(i, j, j') \implies \text{valid}(i, j'),$$

by definition. \square

Lemma 4 Given $i \leq j$, $\text{valid}(i, j)$, and $j' > j$,

$$\begin{aligned} & \text{valid}(i, j') \wedge (\nexists j'' \in (j, j'): \text{valid}(i, j'')) \\ & \implies \text{linked}(i, j, j') \wedge (\exists k \in [i+1, j+1]: \text{valid}(k, j')) \end{aligned}$$

Proof. We need to show that if j' is the smallest index after j such that a valid range $[i, j']$ is formed, then there must be a $k \in [i + 1, j + 1]$ such that $\text{valid}(k, j')$ and $\text{linked}(i, j, j')$. Lemma 1 states that $\text{valid}(i, j')$ implies $\text{linked}(i, j, j')$. Thus, we only need to show the rest.

$\text{valid}(i, j')$ implies that there is a $k' \in [i, j')$ such that $\text{valid}(i, k') \wedge \text{valid}(k' + 1, j')$. If k' were greater than j , then j' would not be the smallest such index, leading to a contradiction. Thus, $k' \in [i, j]$. It follows that $\exists k \in [i + 1, j + 1]: \text{valid}(k, j')$. \square

Lemma 3 and Lemma 4 together indicate a way to find the next valid range efficiently. According to Lemma 4, given $\text{valid}(i, j)$, we need to find the first j' such that $\text{linked}(i, j, j')$ and $\exists k \in [i + 1, j + 1]: \text{valid}(k, j')$. And when we find such a j' , Lemma 3 guarantees $\text{valid}(i, j')$.

We will compute the minimum j' in two steps:

1. Find the smallest $j'' > j$ such that $\text{linked}(i, j, j'')$.
2. Find the smallest $j' \geq j''$ such that $\exists k \in [i + 1, j + 1]: \text{valid}(k, j')$. For j' , $\text{linked}(i, j, j')$ is implied by $\text{linked}(i, j, j'')$ as $j' \geq j''$.

Our algorithm will iterate backwards over i , the starting points of the ranges. During the iteration we will maintain two arrays:

1. `firstEdge`: `firstEdge[j]` stores the smallest $k \in [i, j)$ such that $\text{edge}(k, j)$.
2. `firstValid`: `firstValid[j]` stores the smallest $k \in [i + 1, j]$ such that $\text{valid}(k, j)$.

Given these two arrays, our two queries can be answered with similar operations:

1. Find the smallest $j'' \geq j + 1$ such that `firstEdge[j'']` $\leq j$. Note that finding this edge can be used in cardinality estimation computations as well.
2. Find the smallest $j' \geq j''$ such that `firstValid[j']` $\leq j + 1$.

These queries can be answered in $O(\log n)$ time using a segment tree [WW19]. We will describe an efficient implementation in Sec. 4.5. For now, we abstract away these “find first less than” operations behind a FFLT data structure and provide the algorithm for iterating over ranges in Fig. 11.

In Fig. 11, we further utilize the first connected edge to find connected range pairs starting at $j + 1$, by maintaining a list of connected ranges for each starting point. Since this list is sorted ascending, we can use binary search to find the first connected range containing the end of our first edge.

4.4 Analysis

We will now analyze the time complexity of our algorithm. We assume that the number of nodes in the query graph is n and the number of edges is m . We also define c as the

```

1 def produceValidRangePairs():
2     firstEdge = FFLT(n)
3     firstValid = FFLT(n)
4     connected = [[] for _ in range(n)]
5     for i in range(n - 1, 0, -1):
6         for j in adjList[i] if j > i:
7             firstEdge[j] = i
8             j = i
9             while j < n:
10                # We have found valid range [i, j]
11                firstValid[j] = i
12                connected[i].push(j)
13                jp = firstEdge.query(j + 1, j)
14                it = lower_bound(connected[j + 1], jp)
15                for j2 in connected[j + 1][it:]:
16                    # We have found a connected range pair:
17                    # [i, j] and [j + 1, j2]
18                    yield (i, j, j2)
19                j = firstValid.query(jp, j + 1)

```

Fig. 11: Producing valid ranges using the FFLT data structure: `fflt.query(a, b)` returns the first $k \geq a$ such that `fflt[k] <= b`. We maintain two FFLT arrays `firstEdge` and `firstValid` to find the next valid range efficiently. `firstEdge` is updated once for every edge and `firstValid` is updated once for every valid range. We additionally use the `firstEdge` array to find connected range pairs.

number of valid ranges and p as the number of connected adjacent valid range pairs. Note that $c \geq n$, as every node is a valid range of size 1. With this notation, our algorithm runs in $O((m + c) \log n + p)$ -time since we perform 1 segment tree update per edge, 2 segment tree queries and 1 binary search per valid range, and we iterate over all connected adjacent valid range pairs in constant time.

4.5 “Find First Less Than” Data Structure

To efficiently search for valid ranges, we need a data structure on arrays that can support “find first less than” operations on ranges. The FFLT data structure, based on segment trees [WW19], represents an array that supports the following operations:

1. Initialization: Every value in the represented array starts with ∞ .
2. `fflt[i] = v`: Update the value at index i to v .
3. `fflt.query(a, b)`: Return the smallest $k \geq a$ such that `fflt[k] <= b`.

We can run these operations efficiently using a minimum segment tree data structure. A segment tree allows us to do point updates and range minimum queries in $O(\log n)$ -time. It is structured as a binary tree where each leaf node represents a single element of the array,

and each internal nodes stores the minimum of all its children. We can update a value in $O(\log n)$ -time by traversing the tree from the leaf to the root, recomputing the values on the way using their immediate children. For the query(a , b) operation, we need one upwards and one downwards traversal. While traversing upwards, we try to find the first range with start point greater than equal to a such that the range's minimum is less than or equal to b . Then, we traverse downwards to find the first leaf node that corresponds to a value less than or equal to b . The pseudocode for the query operation is shown in Fig. 12. We further visualize an example query operation in Fig. 13.

```

1 def query(n, tree, a, b):
2     # The start and length of the implicit range
3     st, length = a, 1
4     # Go up the tree
5     ind = a + n
6     while tree[ind] > b:
7         if st + length >= n:
8             # We could not find a value <= b
9             return n
10        if ind & 1:
11            # We are the right child, go to right neighbor of parent
12            st += length
13            ind += 1
14        ind //= 2
15        length *= 2
16    # Go down the tree
17    while ind < n:
18        # Go to left child
19        ind *= 2
20        if tree[ind] > b:
21            # Go to right child
22            ind += 1
23    return ind - n

```

Fig. 12: Query operation for the FFLT data structure. The operation finds the first index $k \geq a$ such that $\text{fflt}[k] \leq b$ in $O(\log n)$ -time. We assume the segment tree represents an array of size n where n is a power of 2 and that this tree is stored implicitly in the tree array such that the root node is at index 1 and the children of node i are at indices $2i$ and $2i + 1$.

[0, 7]: 0							
[0, 3]: 0				[4, 7]: 1			
[0, 1]: 0		[2, 3]: 0		[4, 5]: 2		[6, 7]: 1	
0: 0	1: 1	2: 0	3: 3	4: 2	5: 5	6: 1	7: 7

Fig. 13: The operation query(3, 1) is visualized on the segment tree. The blue-colored cells represent the upward traversal, and the orange-colored cells represent the downward traversal. The operation returns the index 6.

4.6 Linearization Transfer

The extended LinDP algorithm, as implemented in Umbra [NF20], produces linearizations for *each* relation, and computes the optimal parenthesizations for each of these linearizations. However, linearizations of different relations often share a common suffix. We introduce a new technique called *linearization transfer*, which exploits this property to avoid redundant computations. The key idea is that, when a linearization is to be parenthesized, one can reuse the DP-state from the *previous* linearization.

Formally, let L_u and L_v be two linearizations output consecutively by IKKBZ (Sec. 3.2). Let p be the position starting from which L_u and L_v coincide, i.e., $\forall i \geq p: L_u(i) = L_v(i)$. This implies that, once the DP-algorithm ran on L_u , we can *reuse* the DP-state to L_v for all subranges of $[p, n]$. Concretely, the outer iteration starting in Line 8 in Fig. 9 would directly start at index $p - 1$ instead of $n - 1$. The DP table $c[\cdot][\cdot]$ can be reused for larger indices $n - 1 \dots p$ and would need to be reset for the rest of the indices $p - 1 \dots 0$. We visualize this in Fig. 14 for two linearizations L_1 and L_2 . In this case, $p = 4$, so we can reuse the DP-subtable $c[4:][:]$ computed for linearization L_1 in the DP-algorithm on L_2 .

Similarly, the outer iteration starting in Line 5 in Fig. 11 would directly start at index $p - 1$ instead of $n - 1$. For all indices before p , the corresponding entries in the connected arrays are completely cleared. For indices from p onward, all elements in the connected arrays that are smaller than p are removed. This can be done efficiently using binary search since the arrays are sorted. After this adjustment, the smallest remaining element in each connected array is assigned to `firstValid` for that index. Finally, the `firstEdge` array is recomputed for indices from p onward by iterating over the relevant edges. For indices before p , `firstEdge` is reset to infinity.

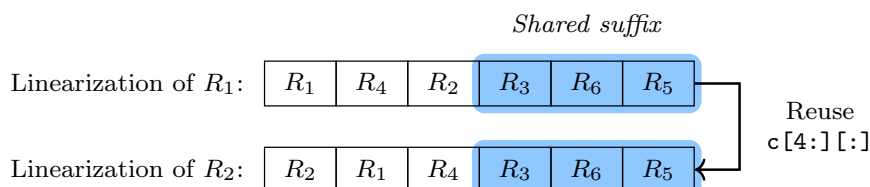


Fig. 14: Two linearizations that share a common suffix. We can *reuse* the DP-subtable $c[4:][:]$.

Optimal Transfer Order. A natural question to ask is whether we could maximize the effect of linearization transfer. Indeed, this is possible, by *sorting* the linearizations based on the suffix. Intuitively, this is simply a lexicographic sorting on the *reversed* linearizations. This order ensures that the length of the shared suffix is maximized across all pairs of consecutive linearizations. As we show experimentally in Sec. 5, enabling linearization transfer in the extended version of linearized dynamic programming further improves its runtime bound across all query graph shapes.

5 Evaluation

In this section, we evaluate *adaptive LinDP*. In particular, we will analyze the impacts of adaptive dynamic programming and linearization transfer on different query structures. We will show that adaptive dynamic programming works especially well in sparse query structures, while linearization transfer is beneficial for dense query structures. Thus, the combination of both techniques is able to adapt to the query structure for a large class of queries and significantly outperform the state of the art. For all the benchmarks, we use a AMD Ryzen 9 5950X CPU. The source code is available [Bi25].

5.1 Query Structures

To test the performance of the various approaches, we generate random query graphs of four different structures: *star*, *chain*, *clique*, and *tree*.

1. A *star* graph is generated by connecting all nodes to the first node.
2. A *chain* graph is generated by connecting each node to the next node.
3. A *clique* graph is generated by connecting each node to all other nodes.
4. A *tree* graph is generated with a *diameter* parameter in the range $[0, 1]$. A *chain* of length $n \cdot \text{diameter}$ is generated from randomly picked nodes. Then, remaining nodes are randomly picked, and connected to a random node in the chain. A tree with a diameter of 0 is close to a *star* graph, while a tree with a diameter of 1 is equivalent to a *chain* graph. For a given diameter, five different trees are generated.

All join selectivities within the query graphs are generated randomly between 0 and 1, rerolled and measured five times.

5.2 DP Algorithms

We compare our improved DP algorithm that only produces valid range pairs with the baseline DP algorithm in Fig. 15. Note that our baseline implementation contains the efficient computation of $\text{linked}(i, k, j)$ as described in Sec. 4.2. With star queries, the baseline DP algorithm is only able to compute the join order for around 20 thousand relations in one second, while our improved DP algorithm from Sec. 4.3 is able compute the join order for 2.5 million relations in the same amount of time. This is due to the fact that the improved DP algorithm has time complexity $\mathcal{O}(n \log n)$ for stars, while the baseline DP algorithm has time complexity $\mathcal{O}(n^2)$. Tree queries (with diameter 0) demonstrate a similar behavior to star queries where the improved DP algorithm is faster by orders of magnitude.

With chain queries, the baseline DP algorithm is only able to compute the join order for around 1.4 thousand relations in one second, while the improved DP algorithm is able

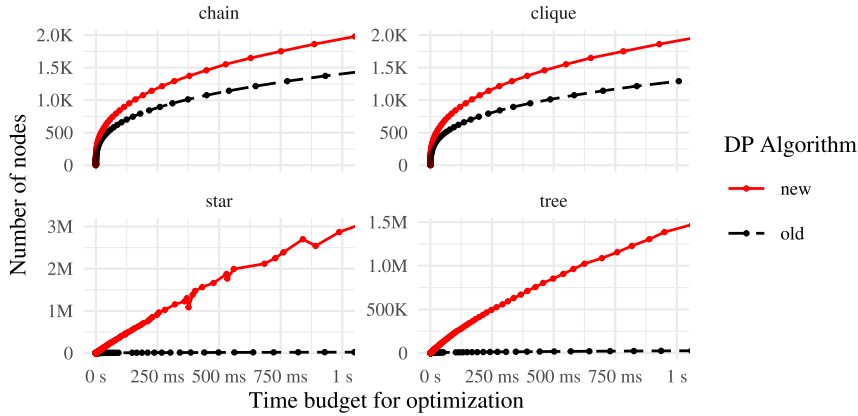


Fig. 15: DP parenthesization algorithms compared on various graph structures. The improved DP algorithm can perform the join order optimization of 2 thousand relations for chains and cliques and over a million relations for stars and trees in one second. For each graph type with a given number of nodes, the median runtime across its variants is shown.

to compute the join order for 1.9 thousand relations in the same amount of time. Even though both algorithms have the same time complexity for chain queries, the improved DP algorithm benefits from the fact that it needs to do fewer checks for validity of range pairs. Clique queries demonstrate a similar behavior to chain queries, but both algorithms are slightly slower compared to chain queries due to the increased number of edges.

5.3 Adaptive LinDP

We evaluate the impact of various optimizations of our approach, adaptive LinDP, in Fig. 16. We vary both the linearization transfer and the adaptive dynamic programming. Linearization transfer is either off, using the original basic IKKBZ order, or sorts the linearizations before applying DP. We also vary the dynamic programming algorithm used, either the baseline DP algorithm or the improved DP algorithm. For all the combinations, we measured the maximum number of nodes that can be optimized under one second for trees of varying diameters. If a tree has a diameter of 0, it is close to a star graph, while a tree with a diameter of 1 is equivalent to a chain graph. We find that, with small diameters, the DP algorithm has the most impact, as most ranges are not valid. The improved DP algorithm is able to optimize around more than twice as many nodes in a second compared to the baseline. With larger diameters, the linearization transfer has the most impact, as the number of valid ranges increases. Turning on linearization transfer results in more than twice the number of nodes optimized in a second. We also find that sorting the linearizations (as explained in Sec. 4.6) has a modest but measurable improvement on the overall performance.

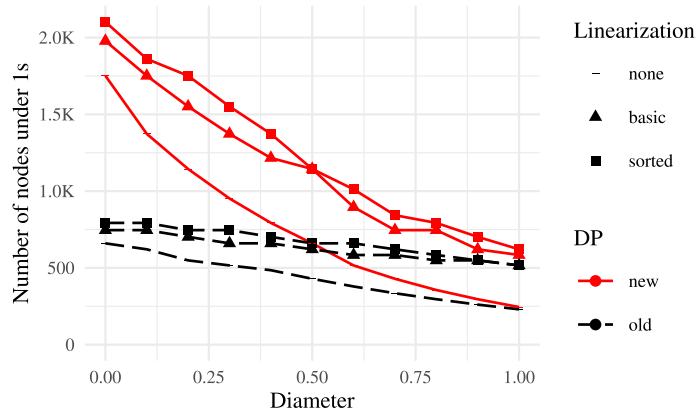


Fig. 16: Maximum number of nodes `LindP` can optimize under 1s for trees of varying diameters. These measurements include the times for the linearization and the parenthesization of each linearization. Linearization transfer has the highest relative impact for high diameters while the improved DP algorithm has the highest relative impact for low diameters.

6 Conclusion

We have presented optimizations for the `LindP` algorithm that improve its performance by orders of magnitude for various queries by adapting its execution to the query structure. The resulting algorithm can provide a high quality join order for millions of relations under a second for all but the most complex queries. This makes our approach a robust candidate for a fallback strategy in query optimizers for queries where exhaustive enumeration of all possible join orders is not feasible.

Future Work. Our approach is currently limited to inner joins with simple predicates, and does not support query graphs hyperedges as used in [MN08; RN19]. In future work, we plan to extend our approach to support such query graphs. Furthermore, since our approach depends on the accuracy of cardinality estimates, we plan to explore integrating our algorithm with techniques designed to improve robustness against estimation errors [BKN24].

Beyond Databases. Our work has implications beyond database join enumeration. Recently, Blacher et al. [B123] introduced a SQL-based view of Einstein summation, also known as `einsum`, widely used as abstraction for tensor operations in deep learning and tensor network frameworks. They ran the generated SQL queries—which could reach hundreds of relations for the SAT expressions of the Anaconda package manager [An24]—on several DBMSes, showing that, in such applications, the query optimizer is the bottleneck. Thus, it is crucial to also scale to large queries appearing in non-database applications. Moreover, `LindP` has been used in tensor contraction ordering, outperforming prior work on tree-shaped tensor network contraction costs [SMM24].

References

- [An24] Anaconda, Inc.: Anaconda Software Distribution, 2024, URL: <https://anaconda.com>.
- [BGJ10] Bruno, N.; Galindo-Legaria, C. A.; Joshi, M.: Polynomial Heuristics for Query Optimization. In (Li, F.; Moro, M. M.; Ghandeharizadeh, S.; Haritsa, J. R.; Weikum, G.; Carey, M. J.; Casati, F.; Chang, E. Y.; Manolescu, I.; Mehrotra, S.; Dayal, U.; Tsotras, V. J., eds.): Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA. IEEE Computer Society, pp. 589–600, 2010, DOI: 10.1109/ICDE.2010.5447916, URL: <https://doi.org/10.1109/ICDE.2010.5447916>.
- [Bi25] Birler, A.: Implementation of Adaptive LinDP, Accessed: 2025-01-07, 2025, URL: <https://github.com/umbra-db/adaptivelindp-btw2025>.
- [BKN24] Birler, A.; Kemper, A.; Neumann, T.: Robust Join Processing with Diamond Hardened Joins. Proc. VLDB Endow. 17 (11), pp. 3215–3228, 2024, DOI: 10.14778/3681954.3681995, URL: <https://www.vldb.org/pvldb/vol17/p3215-birler.pdf>.
- [BI23] Blacher, M.; Klaus, J.; Staudt, C.; Laue, S.; Leis, V.; Giesen, J.: Efficient and Portable Einstein Summation in SQL. Proc. ACM Manag. Data 1 (2), 121:1–121:19, 2023, DOI: 10.1145/3589266, URL: <https://doi.org/10.1145/3589266>.
- [Ch09] Chen, Y.; Cole, R. L.; McKenna, W. J.; Perfilov, S.; Sinha, A.; Jr., E. S.: Partial Join Order Optimization in the Paracel Analytic Database. In (Çetintemel, U.; Zdonik, S. B.; Kossmann, D.; Tatbul, N., eds.): Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009. ACM, pp. 905–908, 2009, DOI: 10.1145/1559845.1559945, URL: <https://doi.org/10.1145/1559845.1559945>.
- [Ch95] Chaudhuri, S.; Krishnamurthy, R.; Potamianos, S.; Shim, K.: Optimizing Queries with Materialized Views. In (Yu, P. S.; Chen, A. L. P., eds.): Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan. IEEE Computer Society, pp. 190–200, 1995, DOI: 10.1109/ICDE.1995.380392, URL: <https://doi.org/10.1109/ICDE.1995.380392>.
- [Co70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13 (6), pp. 377–387, 1970, DOI: 10.1145/362384.362685, URL: <https://doi.org/10.1145/362384.362685>.
- [Di09] Dieu, N.; Dragusanu, A.; Fabret, F.; Llirbat, F.; Simon, E.: 1,000 Tables Inside the From. Proc. VLDB Endow. 2 (2), pp. 1450–1461, 2009, DOI: 10.14778/1687553.1687572, URL: <http://www.vldb.org/pvldb/vol2/vldb09-1077.pdf>.
- [DT07] DeHaan, D.; Tompa, F. W.: Optimal top-down join enumeration. In (Chan, C. Y.; Ooi, B. C.; Zhou, A., eds.): Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007. ACM, pp. 785–796, 2007, DOI: 10.1145/1247480.1247567, URL: <https://doi.org/10.1145/1247480.1247567>.
- [FBN23] Fent, P.; Birler, A.; Neumann, T.: Practical planning and execution of groupjoin and nested aggregates. VLDB J. 32 (6), pp. 1165–1190, 2023, DOI: 10.1007/S00778-022-00765-X, URL: <https://doi.org/10.1007/s00778-022-00765-x>.
- [Fe98] Fegaras, L.: A New Heuristic for Optimizing Large Queries. In (Quirchmayr, G.; Schweighofer, E.; Bench-Capon, T. J. M., eds.): Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings. Vol. 1460. Lecture Notes in Computer Science, Springer, pp. 726–735, 1998, DOI: 10.1007/BFb0054528, URL: <https://doi.org/10.1007/BFb0054528>.

- [FM11] Fender, P.; Moerkotte, G.: A new, highly efficient, and easy to implement top-down join enumeration algorithm. In (Abiteboul, S.; Böhm, K.; Koch, C.; Tan, K., eds.): Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany. IEEE Computer Society, pp. 864–875, 2011, DOI: 10.1109/ICDE.2011.5767901, URL: <https://doi.org/10.1109/ICDE.2011.5767901>.
- [FM12] Fender, P.; Moerkotte, G.: Reassessing Top-Down Join Enumeration. IEEE Trans. Knowl. Data Eng. 24 (10), pp. 1803–1818, 2012, DOI: 10.1109/TKDE.2011.235, URL: <https://doi.org/10.1109/TKDE.2011.235>.
- [Fr24] Franz, M.; Winker, T.; Groppe, S.; Mauerer, W.: Hype or Heuristic? Quantum Reinforcement Learning for Join Order Optimisation. In: Proceedings of the IEEE International Conference on Quantum Computing and Engineering. 2024, URL: <https://arxiv.org/abs/2405.07770>.
- [HD23] Haffner, I.; Dittrich, J.: Efficiently Computing Join Orders with Heuristic Search. Proc. ACM Manag. Data 1 (1), 73:1–73:26, 2023, DOI: 10.1145/3588927, URL: <https://doi.org/10.1145/3588927>.
- [HS82] Hu, T. C.; Shing, M. T.: Computation of Matrix Chain Products. Part I. SIAM J. Comput. 11 (2), pp. 362–373, 1982, DOI: 10.1137/0211028, URL: <https://doi.org/10.1137/0211028>.
- [HS84] Hu, T. C.; Shing, M. T.: Computation of Matrix Chain Products. Part II. SIAM J. Comput. 13 (2), pp. 228–251, 1984, DOI: 10.1137/0213017, URL: <https://doi.org/10.1137/0213017>.
- [IK84] Ibaraki, T.; Kameda, T.: On the Optimal Nesting Order for Computing N-Relational Joins. ACM Trans. Database Syst. 9 (3), pp. 482–502, 1984, DOI: 10.1145/1270.1498, URL: <https://doi.org/10.1145/1270.1498>.
- [KBZ86] Krishnamurthy, R.; Boral, H.; Zaniolo, C.: Optimization of Nonrecursive Queries. In (Chu, W. W.; Gardarin, G.; Ohsuga, S.; Kambayashi, Y., eds.): VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings. Morgan Kaufmann, pp. 128–137, 1986, URL: <http://www.vldb.org/conf/1986/P128.PDF>.
- [KS00] Kossmann, D.; Stocker, K.: Iterative dynamic programming: a new class of query optimization algorithms. ACM Trans. Database Syst. 25 (1), pp. 43–82, 2000, DOI: 10.1145/352958.352982, URL: <https://doi.org/10.1145/352958.352982>.
- [Ma22] Mancini, R.; Karthik, S.; Chandra, B.; Mageirakos, V.; Ailamaki, A.: Efficient Massively Parallel Join Optimization for Large Queries. In (Ives, Z. G.; Bonifati, A.; Abbadi, A. E., eds.): SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022. ACM, pp. 122–135, 2022, DOI: 10.1145/3514221.3517871, URL: <https://doi.org/10.1145/3514221.3517871>.
- [MBL17] May, N.; Böhm, A.; Lehner, W.: SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In (Mitschang, B.; Nicklas, D.; Leymann, F.; Schöning, H.; Herschel, M.; Teubner, J.; Härder, T.; Kopp, O.; Wieland, M., eds.): Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings. Vol. P-265. LNI, GI, pp. 545–563, 2017, URL: <https://dl.gi.de/handle/20.500.12116/656>.
- [MN06] Moerkotte, G.; Neumann, T.: Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In (Dayal, U.; Whang, K.; Lomet, D. B.; Alonso, G.; Lohman, G. M.; Kersten, M. L.; Cha, S. K.; Kim, Y., eds.): Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006. ACM, pp. 930–941, 2006, URL: <http://dl.acm.org/citation.cfm?id=1164207>.

- [MN08] Moerkotte, G.; Neumann, T.: Dynamic programming strikes back. In (Wang, J. T., ed.): Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008. ACM, pp. 539–552, 2008, DOI: 10.1145/1376616.1376672, URL: <https://doi.org/10.1145/1376616.1376672>.
- [MS96] Moerkotte, G.; Scheufele, W.: Constructing Optimal Bushy Processing Trees for Join Queries is NP-Hard, tech. rep., 1996.
- [Ne09] Neumann, T.: Query simplification: graceful degradation for join-order optimization. In (Çetintemel, U.; Zdonik, S. B.; Kossmann, D.; Tatbul, N., eds.): Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009. ACM, pp. 403–414, 2009, DOI: 10.1145/1559845.1559889, URL: <https://doi.org/10.1145/1559845.1559889>.
- [NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. [www.cidrdb.org](http://cidrdb.org), 2020, URL: <http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf>.
- [NR18] Neumann, T.; Radke, B.: Adaptive Optimization of Very Large Join Queries. In (Das, G.; Jermaine, C. M.; Bernstein, P. A., eds.): Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. ACM, pp. 677–692, 2018, DOI: 10.1145/3183713.3183733, URL: <https://doi.org/10.1145/3183713.3183733>.
- [OL90] Ono, K.; Lohman, G. M.: Measuring the Complexity of Join Enumeration in Query Optimization. In (McLeod, D.; Sacks-Davis, R.; Schek, H., eds.): 16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings. Morgan Kaufmann, pp. 314–325, 1990, URL: <http://www.vldb.org/conf/1990/P314.PDF>.
- [Re24] van Renen, A.; Horn, D.; Pfeil, P.; Vaidya, K. E.; Dong, W.; Narayanaswamy, M.; Liu, Z.; Saxena, G.; Kipf, A.; Kraska, T.: Why TPC is not enough: An analysis of the Amazon Redshift fleet. In: VLDB 2024. 2024, URL: <https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet>.
- [RN19] Radke, B.; Neumann, T.: LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins. In (Grust, T.; Naumann, F.; Böhm, A.; Lehner, W.; Härder, T.; Rahm, E.; Heuer, A.; Klettke, M.; Meyer, H., eds.): Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings. Vol. P-289. LNI, Gesellschaft für Informatik, Bonn, pp. 57–76, 2019, DOI: 10.18420/BTW2019-05, URL: <https://doi.org/10.18420/btw2019-05>.
- [Se79] Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G.: Access Path Selection in a Relational Database Management System. In (Bernstein, P. A., ed.): Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1. ACM, pp. 23–34, 1979, DOI: 10.1145/582095.582099, URL: <https://doi.org/10.1145/582095.582099>.
- [SK24] Stoian, M.; Kipf, A.: DPconv: Super-Polynomially Faster Join Ordering, 2024, arXiv: 2409.08013 [cs.DB], URL: <https://arxiv.org/abs/2409.08013>.
- [SMK97] Steinbrunn, M.; Moerkotte, G.; Kemper, A.: Heuristic and randomized optimization for the join ordering problem. The VLDB journal 6, pp. 191–208, 1997.
- [SMM24] Stoian, M.; Milbradt, R. M.; Mendl, C. B.: On the Optimal Linear Contraction Order of Tree Tensor Networks, and Beyond. SIAM Journal on Scientific Computing 46 (5), B647–B668, 2024, DOI: 10.1137/23M161286X, URL: <https://doi.org/10.1137/23M161286X>.

- [SSM23] Schönberger, M.; Scherzinger, S.; Mauerer, W.: Ready to Leap (by Co-Design)? Join Order Optimisation on Quantum Hardware. *Proc. ACM Manag. Data* 1 (1), 92:1–92:27, 2023, DOI: 10.1145/3588946, URL: <https://doi.org/10.1145/3588946>.
- [STM23] Schönberger, M.; Trummer, I.; Mauerer, W.: Quantum-Inspired Digital Annealing for Join Ordering. *Proc. VLDB Endow.* 17 (3), pp. 511–524, 2023, URL: <https://www.vldb.org/pvldb/vol17/p511-schonberger.pdf>.
- [Sw89] Swami, A. N.: Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In (Clifford, J.; Lindsay, B. G.; Maier, D., eds.): *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, USA, May 31 - June 2, 1989. ACM Press, pp. 367–376, 1989, DOI: 10.1145/67544.66961, URL: <https://doi.org/10.1145/67544.66961>.
- [TK17] Trummer, I.; Koch, C.: Solving the Join Ordering Problem via Mixed Integer Linear Programming. In (Salihoglu, S.; Zhou, W.; Chirkova, R.; Yang, J.; Suciu, D., eds.): *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017*, Chicago, IL, USA, May 14-19, 2017. ACM, pp. 1025–1040, 2017, DOI: 10.1145/3035918.3064039, URL: <https://doi.org/10.1145/3035918.3064039>.
- [VM96] Vance, B.; Maier, D.: Rapid Bushy Join-order Optimization with Cartesian Products. In (Jagadish, H. V.; Mumick, I. S., eds.): *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Quebec, Canada, June 4-6, 1996. ACM Press, pp. 35–46, 1996, DOI: 10.1145/233269.233317, URL: <https://doi.org/10.1145/233269.233317>.
- [Wi23] Winker, T.; Çalikyilmaz, U.; Gruenwald, L.; Groppe, S.: Quantum Machine Learning for Join Order Optimization using Variational Quantum Circuits. In (Groppe, S.; Gruenwald, L.; Hsu, C., eds.): *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE 2023*, Seattle, WA, USA, 18 June 2023. ACM, 5:1–5:7, 2023, DOI: 10.1145/3579142.3594299, URL: <https://doi.org/10.1145/3579142.3594299>.
- [WW19] Wang, L.; Wang, X.: A Simple and Space Efficient Segment Tree Implementation. *MethodsX* 6, pp. 500–512, 2019, ISSN: 2215-0161, DOI: <https://doi.org/10.1016/j.mex.2019.02.028>, URL: <https://www.sciencedirect.com/science/article/pii/S2215016119300391>.
- [Ya80] Yao, F. F.: Efficient Dynamic Programming Using Quadrangle Inequalities. In (Miller, R. E.; Ginsburg, S.; Burkhard, W. A.; Lipton, R. J., eds.): *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, April 28-30, 1980, Los Angeles, California, USA. ACM, pp. 429–435, 1980, DOI: 10.1145/800141.804691, URL: <https://doi.org/10.1145/800141.804691>.