DPconv: Super-Polynomially Faster Join Ordering

Mihail Stoian
University of Technology Nuremberg
mihail.stoian@utn.de

Andreas Kipf
University of Technology Nuremberg
andreas.kipf@utn.de

ABSTRACT

We revisit the join ordering problem in query optimization. The standard exact algorithm, `DPccp`, has a worst-case running time of $O(3^n)$. This is prohibitively expensive for large queries, which are not that uncommon anymore. We develop a new algorithmic framework based on subset convolution. `DPconv` achieves a super-polynomial speedup over `DPccp`, breaking the $O(3^n)$ time-barrier for the first time. We show that the framework instantiation for the $C_{\max}$ cost function is up to 30x faster than `DPccp` for large clique queries.

1. INTRODUCTION

The query optimizer is the heart of any relational database system. One of the fundamental tasks of the query optimizer is join ordering. The problem is to reorder the joins, so that the query execution time is minimized. To this end, one introduces a cost model that acts as proxy for the actual execution time. Since the costs are directly reflected in the query execution time, optimal or near-optimal join orders are indispensable for the overall performance. However, the problem is inherently NP-hard [18]. This means that, unless P = NP, one has to resort to the exponential (exact) algorithm for small queries and to greedy strategies otherwise.

Motivation & Research Question. In a seminal work, Selinger et al. [38] introduced the first dynamic program to (exactly) optimize the ordering problem. The key observation is that the optimal solution S^* for a set of relations P, called the problem, satisfies Bellman's optimality principle [1], namely that S^* is computed from two disjoint subproblems P_1 and P_2, with optimal solutions S_1^* and S_2^*, respectively. The naive algorithm, `DPsize`, runs in $O(4^n)$-time, which can be reduced to $O(3^n)$ by a careful traversal of the subsets of a given set, algorithm known as `DPsub` [43, 42].

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is a minor revision of the paper "DPconv: Super-Polynomially Faster Join Ordering" published in Proc. ACM Manag. Data, Vol. 2, No. 6 (SIGMOD), Article 234 (December 2024). DOI: https://doi.org/10.1145/3698809

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Copyright 2025 ACM 0001-0782/24/0X00 ...$5.00.

$$\mathrm{DP}[R_1R_2R_3R_4] = \min \begin{cases} \mathrm{DP}[R_1] + \mathrm{DP}[R_2R_3R_4] \\ \mathrm{DP}[R_2] + \mathrm{DP}[R_1R_3R_4] \\ \mathrm{DP}[R_3] + \mathrm{DP}[R_1R_2R_4] \\ \mathrm{DP}[R_4] + \mathrm{DP}[R_1R_2R_3] \\ \mathrm{DP}[R_1R_2] + \mathrm{DP}[R_3R_4] \\ \mathrm{DP}[R_1R_3] + \mathrm{DP}[R_2R_4] \\ \mathrm{DP}[R_1R_4] + \mathrm{DP}[R_2R_3] \end{cases}$$

$$\overset{V := R_1R_2R_3R_4}{=} \min_{\varnothing \subset S \subset V} \left(\mathrm{DP}[S] + \mathrm{DP}[V \setminus S]\right)$$

Subset Convolution

Figure 1: How join ordering dynamic programming algorithms, e.g., `DPsub`, are implicitly using subset convolution. However, they are computing it *naively*. `DPconv` instead uses a highly tuned implementation of *fast subset convolution*.

Later, Moerkotte and Neumann [26] showed that one can obtain an improved algorithm if one disallows cross-products, namely by considering the connectivity structure of the underlying query graph (the algorithm was later extended to query hypergraphs [27]). Their algorithm, `DPccp`, achieves the lower bound on the number of connected complement pairs which any dynamic program needs to traverse, as shown by Ono and Lohman [31]. Recently, Haffner and Dittrich [15] proved that join ordering reduces to computing shortest paths in an exponential-size graph in which the vertices are relation subsets. Their reduction enables the use of well-known speedups via heuristic search, as known from the A^* algorithm [16]. However, while their average-case running time beats that of `DPccp`, the worst-case running time still remains $O(3^n)$. The $O(3^n)$-time bottleneck leads us to our main question:

Is there a way to break the seemingly unyielding $O(3^n)$-time barrier?

Surprisingly, there is. To this end, consider Fig. 1, in which we show how the standard join ordering dynamic programming algorithms `DPsub` [43, 42] and `DPccp` [26] optimize the full set of relations $V = \{R_1, R_2, R_3, R_4\}$. Simply put, the algorithm iterates over all possible ways to split the original set V into two subsets. *This is exactly a subset convolution.* However, all current join ordering algorithms, `DPsize`, `DPsub`, and `DPccp`, perform it naively, i.e., the expression is evaluated as is. Fortunately for our community, research in algorithm design has led to a *fast* subset convolution [2].

Intuitively, fast subset convolution no longer naively enumerates subsets, but instead uses an FFT-inspired strategy that avoids redundant computational steps in the naive evaluation.

In this work, we develop a new exact algorithmic framework based on fast subset convolution that has a superpolynomial speedup over `DPccp` / `DPsub`, breaking the long-standing $O(3^n)$ time-barrier for the first time. We instantiate the framework for two well-studied cost functions, C_{out} and $C_{\max}$, which guarantee time- and space-optimality of query execution, respectively. Namely, the latter minimizes the sum of the intermediate join sizes, while the former minimizes the largest intermediate one. This results in an $O(2^n n^2 Wn \log Wn)$-time algorithm for C_{out}, which is $O^*(2^n)$ when the largest join cardinality W is polynomial in n,[1] and an $O(2^n n^3)$-time algorithm for $C_{\max}$; the latter running time is independent of W. At a practical level, we show that the instantiation for $C_{\max}$ is up to 30x faster than the classic algorithm for clique queries of 17 or more relations.

The framework also allows to further reduce the optimization time for C_{out}, using an $(1+\varepsilon)$-approximation algorithm, to $O^*(2^{3n/2}/\sqrt{\varepsilon})$-time. Unlike our *exact* algorithm for the same cost function, the running time of this algorithm is independent of W, making it attractive for queries with huge join cardinalities.

Contributions. In this shortened article, we will focus on the following two key contributions:

1. We introduce a new exact algorithmic framework based on subset convolution which breaks the long-standing time-barrier of $O(3^n)$ for the first time.

2. We provide a practical instantiation of the framework for $C_{\max}$, achieving an $O(2^n n^3)$-time algorithm.

Organization. The rest of the paper is organized as follows: First, in §2, we formalize the problem of join ordering and that of fast subset convolution. Then, we describe the machinery behind fast subset convolution in §3, and, in §4, we introduce `DPconv` along with the novel connection between join ordering and subset convolution. As part of §5, we provide a simple and *practical* algorithm for $C_{\max}$, which we evaluate in §6. We then outline related work in §7, provide a discussion in §8, and finally conclude in §9.

2. BACKGROUND

In this section, we formalize both problems, namely join ordering and subset convolution.

2.1 Query Graph

Let $\mathcal{D} = \{R_1, \ldots, R_n\}$ be a database that contains n relations. A select-project-join query $\mathcal{Q}$ is defined as

$$\mathcal{Q} = \Pi_A(\sigma_P(R_1 \times \ldots \times R_n)), \tag{1}$$

where P is the conjunction of predicates that can be both join predicates, i.e., $R_i.a = R_j.b$, and selection predicates, i.e., $R_i.a = \mathit{const}$, and A is the list of attributes required to appear in the output. The operators Π, σ, and $\times$ are the projection, selection and cross-product operators, respectively, as defined in the relational algebra [6].

[1] The notation O^* hides polynomial factors in the query size.

We can model a query as a *query graph* $Q = (V, E)$, where the vertex set V corresponds to the set of relations $\{R_i\}_{i \in [n]}$ of the query and the edge set $E = \{\{R_u, R_v\} \mid R_u, R_v \in V\}$ corresponds to the join predicates (called join edges in the sequel). Intuitively, a query can be evaluated by repeatedly joining two relations and replacing one of them with their join. Another prominent way of executing joins by worst-case optimal joins, which are not necessarily *binary* joins anymore [30]. In this work, we only concentrate on query optimization of binary joins. In this case, the order in which the joins are performed can be represented by a (binary) *join tree*, where the leaf nodes are the relations and the inner nodes are the corresponding joins.

2.2 Cost Function

To optimize the join order, one introduces a cost function $\mathcal{C}$ which best models the query execution time. The goal is to minimize the cost function among all possible join trees. Due to the binary structure of a join tree, the cost function can be computed recursively along a join tree $\mathcal{T}$, as follows:

$$\mathcal{C}(\mathcal{T}) = \begin{cases} 0, & \text{if } \mathcal{T} \text{ is a single relation} \\ c(T) \otimes \mathcal{C}(\mathcal{T}_1) \otimes \mathcal{C}(\mathcal{T}_2), & \text{if } \mathcal{T} = \mathcal{T}_1 \bowtie \mathcal{T}_2, \end{cases} \tag{2}$$

where c is the join cardinality function defined on sets of relations, $\otimes$ is an operator that combines the cost-values, T is the set of relations spanned by the join tree $\mathcal{T}$,[2] and $\mathcal{T}_1$ and $\mathcal{T}_2$ are the left and right join subtrees of $\mathcal{T}$, respectively.

Let us instantiate Eq. (2) for two cost functions, C_{out} and $C_{\max}$, which guarantee time- and space-optimality of the query execution, respectively:

$$C_{\text{out}}(T) = c(T) + C_{\text{out}}(\mathcal{T}_1) + C_{\text{out}}(\mathcal{T}_2), \tag{3}$$

$$C_{\max}(T) = \max\{c(T), C_{\max}(\mathcal{T}_1), C_{\max}(\mathcal{T}_2)\}. \tag{4}$$

We can observe that the "$\otimes$" operator from above has been substituted by "+" and "max", respectively.

2.3 Join Ordering and Dynamic Programming

By Bellman's optimality principle [1], the problem of finding the optimal join tree $\mathcal{T}^*$ amounts to finding the optimal *split* of a set of relations S into two disjoint sets S_1 and S_2, i.e., $S_1 \cap S_2 = \varnothing$ and $S = S_1 \cup S_2$. Consequently, given a cost function $\mathcal{C}$, the problem can be optimized by the following dynamic programming (DP) recursion, which closely follows the definition of $\mathcal{C}$:

$$\text{DP}(S) = \begin{cases} 0, & \text{if } |S| = 1 \\ c(S) \otimes \min\limits_{\varnothing \subset T \subset S} (\text{DP}(T) \otimes \text{DP}(S \setminus T)), & \text{otherwise.} \end{cases} \tag{5}$$

Indeed, this is the idea explored by Selinger and the subsequent work [38, 43, 42, 26]. In particular, `DPccp` [26] implements the recursion by considering only sets of relations that induce a connected subgraph; for clique queries, `DPsub` and `DPccp` are both exactly the recursion above. As motivated in Fig. 1, Eq. (5) is a subset convolution. All previous algorithms evaluate it in the naive way, which takes $O(3^n)$-time. `DPconv` speeds up its computation by employing fast subset convolution [2].

[2] Having a separate notation for the join tree and the set of relations it spans will prove useful in the following sections.

2.4 Subset Convolution

Subset convolution is one of the important tools in the field of exact algorithms [14, 7]. Its fast counterpart, called Fast Subset Convolution (FSC) [2], represented a breakthrough in the field by reducing the running time from the straightforward $O(3^n)$ to a non-trivial $O(2^n n^2)$.

Dynamic Programming Speedups. The main application of FSC is the speedup of several dynamic programming recursions of well-known NP-hard problems, such as the Steiner tree problem [9] and min-cost k-coloring [7]. While these problems may seem foreign to our research area, there is a striking similarity between the dynamic programming recursion of these problems and that of the join ordering problem. Indeed, they all implicitly use a subset convolution. Through our work, join ordering is now becoming part of this family of problems [37, 3, 7, 9, 32].

While our main result mostly uses FSC in a black-box manner, we present the full machinery behind it in §3. Note that for an efficient implementation of the dynamic programs, we refer the interested reader to Ref. [40, Sec. 5], as part of which we shaved a linear factor from the running time of generic FSC-based dynamic programs, as well as several constant factors hidden behind the running time's big-O notation; in the sequel, we refer to it as LayeredDP.

Key Idea. Let us first gain an intuition about how subset convolution works at a high level. First, note that the DP-table is by definition a set function: It maps subsets of relations to their corresponding costs. The usual way to refer to a subset structure is by a subset lattice, in our case of order n, since we have n relations. This leads to the following setting: Let f and g be two set functions on the subset lattice of order n, their subset convolution in the $(+, \times)$ ring is defined for *all* $S \subseteq [n] := \{1, \ldots, n\}$ by

$$h(S) = (f * g)(S) = \sum_{T \subseteq S} f(T) g(S \setminus T).$$

Next we describe the low-level workings of FSC.

3. FAST SUBSET CONVOLUTION

We take a closer look at FSC from a practical perspective. In the following, we adopt the notation from the Parameterized Algorithms book [7], since it has established itself in the literature compared to that of Björklund et al. [2].

3.1 Zeta Transform

A fundamental operation in FSC is the zeta transform, defined as

$$(\zeta f)(S) = \sum_{T \subseteq S} f(T), \tag{6}$$

for any $S \subseteq [n]$. That is, the zeta transform sums f at all subsets of S. Naively, this can be computed in $O(3^n)$-time for all $S \subseteq [n]$. However, we can compute it in $O(2^n n)$-time by observing that we can reuse the computation done for subsets. We detail this in §3.4.

3.2 Ranked Convolution

Given ζf and ζg, the zeta transform of the actual convolution $h = f * g$, i.e., ζh, can now be computed point-wise. To this end, Björklund et al. [2] employ a *ranked* convolution. Formally,

$$(\zeta h)(S, r) = \sum_{d=0}^{r} (\zeta f)(S, d)(\zeta g)(S, r - d), \tag{7}$$

for any $S \subseteq [n]$, where $|S| = r$. Thus, we have to apply a zeta transform for *each* rank, i.e., for each cardinality in $\{0, \ldots, n\}$. The ranked convolution can then be computed naively in $O(2^n n^2)$, as for each rank r we need to iterate over all $d \leq r$.

3.3 Möbius Transform

To obtain the actual convolution, one applies the Möbius transform rank-wise. The Möbius transform is indeed the *inverse* of the zeta transform, i.e., $\zeta\mu = \mu\zeta = \text{id}$, and is defined for any $S \subseteq [n]$ as

$$(\mu f)(S) = \sum_{T \subseteq S} (-1)^{|T|} f(T). \tag{8}$$

A full-fledged example of FSC is shown in §3.5 and its associated Fig. 2.

3.4 Implementation

Zeta Transform. A naive evaluation of Eq. (6) leads to an $O(3^n)$-time algorithm, as for each subset we are to sum up along all its subsets. However, there is a faster way computing it, commonly referred to as Yates' algorithm [46]. Define $\hat{f}_0(S) = f(S)$ for all $S \subseteq [n]$, and then iterate for all $j = 1, 2, \ldots, n$ and $S \subseteq [n]$ as follows [2]:

$$\hat{f}_j(S) = \begin{cases} \hat{f}_{j-1}(S) & \text{if } j \notin S, \\ \hat{f}_{j-1}(S \setminus \{j\}) + \hat{f}_{j-1}(S) & \text{if } j \in S. \end{cases} \tag{9}$$

By induction, one can show that $\hat{f}_n(S) = (\zeta f)(S)$ for all $S \subseteq [n]$. The computation of Eq. (9) takes $O(2^n n)$ operations, as for each subset S we need to iterate over its elements. Lst. 1 shows an implementation of Eq. (9).[3]

```
1 zeta(f):
2   for (d = 0; d != n; ++d):
3     for (S = 0; S != 2**n; ++S):
4       if S & 2**d:
5         f[S] += f[S ^ 2**d]
```

Listing 1: Zeta transform

Möbius Transform. The Möbius transform can be computed in a similar way. Define $\check{f}_0(S) = f(S)$ for all $S \subseteq [n]$, and then evaluate the following recursion [2]:

$$\check{f}_j(S) = \begin{cases} \check{f}_{j-1}(S) & \text{if } j \notin S, \\ -\check{f}_{j-1}(S \setminus \{j\}) + \check{f}_{j-1}(S) & \text{if } j \in S. \end{cases} \tag{10}$$

Then one can show that $\check{f}_n(S) = (\mu f)(S)$ for all $S \subseteq [n]$ and the computation happens in $O(2^n n)$ operations as well.

A sketch of the entire FSC algorithm is shown in Lst. 3. We use the already-established Python's slicing notation ":" to denote an entire axis of the array, in our case, indexed by bitsets corresponding to the actual sets of relations. We next show a working example.

[3] G. Moerkotte made us aware that the zeta transform can, intriguingly, be implemented slightly faster using the complete "design matrix" [17], i.e., the binary matrix that encodes the subset relation.

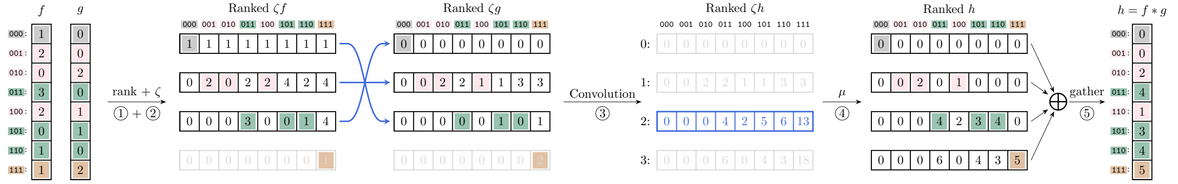

Figure 2: Visualizing the fast subset convolution (FSC), outlined in Lst. 3: We ① rank the set functions f and g and ② apply the zeta transform to obtain ζf and ζg, respectively. Then, we ③ perform the ranked convolution between ζf and ζg and ④ apply the Möbius transform to obtain the ranked h.Finally, we ⑤ reconstitute $h = f * g$, the actual subset convolution. We highlight in (this) color the steps needed to compute the rank "slice" $(\zeta h)(:, 2)$ of ζh during ranked convolution (as in §3.2). Intuitively, we need to sum up the dot products between the corresponding slices, i.e., $(\zeta f)(:, 0)$ with $(\zeta g)(:, 2)$, $(\zeta f)(:, 1)$ with $(\zeta g)(:, 1)$, and $(\zeta f)(:, 2)$ with $(\zeta g)(:, 0)$.

```
 1 FSC(f, g):
 2   for (r in [0, n]):
 3     // Rank f and g
 4     for (S with |S| = r):              ①
 5       f[S, r] = f[S]
 6       g[S, r] = g[S]
 7
 8     // Zeta transform
 9     zf[S, r] = zeta(f[:, r])            ②
10     zg[S, r] = zeta(g[:, r])
11
12     // Ranked convolution
13     for (d in [0, r])                   ③
14       zh[:, r] += zf[:, d] * zg[:, r - d]
15
16     // Moebius transform
17     h[:, r] = mu(zh[:, r])              ④
18
19   // Reconstitute h
20   for (S in [0, 2**n]):                 ⑤
21     h[S] = h[S, |S|]
```

Figure 3: Fast subset convolution, visualized in Fig. 2.

Running Time. Let us calculate the total running time of FSC: The n zeta and Möbius transforms take in total $O(2^n n^2)$-time, while the rank convolution itself takes $O(2^n n^2)$-time.

3.5 Example

To facilitate the understanding of how fast subset convolution works, we provide a working example in Fig. 2. In particular, we visualize the steps of Lst. 3. Our example considers two set functions, f and g, of size 8, i.e., a subset lattice of size 3. Note that we have combined steps ① and ② in Fig. 2 into a single step.

① **Rank.** In the first step, we create as many rank "slices" as there are set cardinalities; in our case, there are 4 rank slices in total. Initially, they all contain only the values corresponding to the positions of the same cardinality. For instance, the slice corresponding to rank 2 is initially comprised of the values at positions `011`, `101`, and `110`. Accordingly, these positions and values are displayed in the same color.

② **Applying Zeta.** Once we created the rank slices, we can now apply the zeta transform to them. Recall its definition in Eq. (6): For each set S, we sum all the values of f (and analogously for g) of at the indices of S's subsets. To show this, consider the same rank slice 2 of ζf: The value at `111` – which is 4 – is the sum of $3 + 1$. Similar in the rank slice 1 of ζg: The value at $(\zeta g)(\texttt{111}, 1)$ is made up of the non-zero values $g(\texttt{010})$ and $g(\texttt{100})$.

③ **Ranked Convolution.** Once both ranked ζf and ζg have been computed, we can run the (ranked) convolution between them, as described in Eq. (7). We visualize the steps for computing the rank slice 2 of ζh in Fig. 2, namely: The colored arrows connecting $(\zeta f)(:, 0)$ with $(\zeta g)(:, 2)$, $(\zeta f)(:, 1)$ with $(\zeta g)(:, 1)$, and $(\zeta f)(:, 2)$ with $(\zeta g)(:, 0)$ show that we need to multiply these rank slices to obtain $(\zeta h)(:, 2)$. As pointed out in Eq. (7), we simply perform a dot product between these and sum up the results. For instance, to obtain $(\zeta h)(\texttt{111}, 2)$, we need to perform the following calculation: $1 \cdot 1 + 4 \cdot 3 + 4 \cdot 0 = 13$.

④ **Applying Möbius.** To obtain the actual "ranked" h, we have to apply the Möbius transform onto ranked ζh. As explained in §3.3, the Möbius transform is the *inverse* of the zeta transform. Once this is done, the next paragraph explains how to obtain the final subset convolution result, h. The Möbius transform is applied as in Eq. (8), namely we consider all the subsets of a set S and *subtract* the values where the subset cardinality is odd, and *add* those for even cardinality. For instance, $h(\texttt{111}, 2)$ is computed as follows: The values $\zeta h(\texttt{100}, 2), \zeta h(\texttt{111}, 2)$ are at odd cardinalities, so we subtract their values, while $\zeta h(\texttt{011}, 2), \zeta h(\texttt{101}, 2)$, and $\zeta h(\texttt{110}, 2)$ are at even cardinalities, so we add them. In total, this results in $-2-13+4+5+6 = 0$, which is exactly $h(\texttt{111}, 2)$.[4]

⑤ **Gather.** Finally, once the ranked h has been fully computed by applying the Möbius transform to ζh, we can obtain the final h by taking a reverse process to step ①: Instead of scattering the set functions to rank slices, we now gather the rank slices into one set function. This is done by simply taking the positions from the corresponding rank slice and putting these into h; this is also highlighted by the corresponding colors. For instance, to collect the positions `011`, `101`, and `110`, we take them from the rank slice 2, since all these subsets have cardinality 2.

This concludes our example. In the following, we relate the join ordering problem to fast subset convolution for the first time, and provide a unified framework that can be instantiated for several cost functions.

[4] We corrected the calculation after receiving an anonymous feedback.

Algorithm 1: `DPconv`: Using fast subset convolution (FSC) to gradually optimize the dynamic programming table.

1: **Input:** Query graph $Q = (V, E)$, cardinality function c
2: **Output:** Optimal cost value w.r.t. $\mathcal{C}$
3: $\text{DP}[\varnothing] \leftarrow +\infty$
4: $\text{DP}[\{R_i\}] \leftarrow 0, \forall R_i \in V$
5: **for each** k **in** $2, \ldots, |V|$ **do**
6: $\quad \text{DP}' \leftarrow \text{FSC}_{(\min,\otimes)}(\text{DP}, \text{DP})$
7: $\quad \text{DP}[S] \leftarrow \text{DP}'[S] \otimes c(S), \forall S \text{ s.t. } |S| = k$
8: **end for**
9: **return** $\text{DP}[V]$

4. OUR FRAMEWORK

Let us consider the optimization of an *arbitrary* cost function $\mathcal{C}$ in its associated $(\min, \otimes)$ semi-ring under a generic framework.[5] We will then instantiate the framework for C_{out} and $C_{\max}$, respectively.

4.1 Join Ordering Meets Subset Convolution

The key observation behind our results is the (now trivial) observation that the definition of DP-recursion, Eq. (5), is similar to that of subset convolution. In particular, we show that join ordering falls into the category of dynamic programs which fast subset convolution has already been applied to. In our specific context, there are a few (minor) issues that need to be addressed for FSC to be applicable, issues that have also been considered by Björklund et al. [2] for other problems, namely:

(i) The dynamic program DP, Eq. (5), is defined recursively.

(ii) The subset T of S in the same Eq. (5) must not take $\varnothing$ nor S as value.

Overview. Both issues are resolved by a simple technique: We apply FSC layer-wise, i.e., we optimize sets of size 2 first, then those of size 3, and so on – this is what we call a *layer*. Specifically, at each layer k, since the DP-table has been computed for layers $k' < k$, we can directly optimize $\text{DP}[S]$ for all S with $|S| = k$ by a call to FSC. To alleviate issue (ii), we set $\text{DP}[\varnothing]$, i.e., the DP-cell representing the empty set of relations, to $+\infty$. Since FSC is called n times, the total optimization time adds up to $O(2^n n^3 \tau_{\mathcal{C}})$, where the function $\tau_{\mathcal{C}}$ is tailored to the specific cost function $\mathcal{C}$ and accounts for the time overhead of semi-ring operations. We will cover the exact expressions of $\tau_{\mathcal{C}}$ for individual cost functions in the following sections (see §4.3 for C_{out} and §4.4 for $C_{\max}$).

Note that our improved LayeredDP [40, Sec. 5] shaves an $O(n)$-factor off the running time, resulting in a total running time of $O(2^n n^2 \tau_{\mathcal{C}})$ for a cost function $\mathcal{C}$. We skip the details and refer the interested reader to the full paper.

Pseudocode. In Alg. 1, we outline the pseudocode behind our framework `DPconv`. It takes the query graph Q and the join cardinality function c as input and outputs the optimal cost value w.r.t. the specific cost function $\mathcal{C}$ which the $(\min, \otimes)$ semi-ring corresponds to.

The algorithm first optimizes the base cases, namely for the empty set of relations and for all sets containing only one relation. The former are initialized with $+\infty$, as argued above, and the latter with 0, cf. Eq. (5). Then, at each layer k, we optimize the subsets of size k by calling FSC on the current state of the DP-table (line 6) and then update the values with the join cardinalities of the subsets (line 7). To this end, note that Alg. 1 can optimize for cross-products out of the box: We simply need to also use the cardinalities of all cross-products in c. The running time remains naturally the same. Finally, we return the optimal cost represented by $\text{DP}[V]$.

[5] We ask the reader to not worry about the semi-ring notation.

Algorithm 2: `BuildJoinTree`: Recursively extracting the optimal bushy join tree from the DP-table

1: **Input:** Subset of relations S, DP-table
2: **Output:** The optimal bushy join tree
3: **if** $|S| = 1$ **return** S **end if**
4: **for each** $\varnothing \subset T \subset S$ **do**
5: $\quad$ **if** $c(S) \otimes \text{DP}(T) \otimes \text{DP}(S \setminus T) = \text{DP}(S)$ **then**
6: $\quad\quad$ **return** (`BuildJoinTree`(T), `BuildJoinTree`$(S \setminus T)$)
7: $\quad$ **end if**
8: **end for**

Join Tree Extraction. Note that unlike previous algorithms, Alg. 1 does not maintain an OPT-table that stores the optimal split for each subset S. This is because FSC itself does not keep track of this information during its execution. In contrast, after the DP-table is fully-optimized, we can extract the optimal join tree from the DP-table itself, as outlined in Alg. 2. Specifically, for each set S, we find the subset T that was *intrinsically* used in FSC to optimize S, i.e., $\text{DP}[S] = \text{DP}[T] + \text{DP}[S \setminus T]$. Since there are at most n levels of recursion, the worst-case running time for Alg. 2 reads $O(2^n n)$.

Cross-Products. While allowing cross-products can lead to better overall costs [31, 33], the search space increases exponentially [31]. A prominent way to deal with cross-products is to heuristically insert them when they are guaranteed to be beneficial [31]; this tends to be the case when the estimated cardinality of the input is small enough [23, 33].

A natural question is whether `DPconv` could also support the optimization of cross-products, and whether this particular optimization would take more time than previously specified. Similar to `DPsub` [43, 42], we can use the cardinalities of the cross-products directly in c (line 7, Alg. 1). This means that `DPconv` can optimize for cross-products for both cost functions without any overhead.

We now come to the embedding technique, motivated and mentioned in §2.4, that helps us leverage the running time of the fast subset convolution to our employed semi-rings.

4.2 Embedding Technique

The embedding technique maps the values of the set functions to monomials and then runs the fast subset convolution algorithm in the $(+, \times)$ ring. The convolution values can then be read from the resulting polynomials. To see why this works, consider the functions `[2, 1, 3, 4]` and `[5, 0, 1, 2]`. When we embed these values to monomials, we obtain $[x^2, x^1, x^3, x^4]$ and $[x^5, x^0, x^1, x^2]$, respectively. Thus, by running their subset convolution $[x^2, x^1, x^3, x^4] * [x^5, x^0, x^1, x^2]$, we can retrieve the final values as follows: Consider the value at `001`, which is $x^{2+0} + x^{1+5}$. Note that multiplication between monomials is simply an addition at

the exponent level, while the minimum—in our case, $\min\{2+0, 1+5\}$—is represented by the smallest exponent in the resulting polynomial.

Representation. To allow for a seamless instantiation of our framework for other cost functions, we represent the polynomials in *coefficient form*, i.e., pairs of exponents and their associated coefficients. For instance, we represent $2x + 3x^4$ as $\{(1, 2), (4, 3)\}$.

Limitation. The core limitation of the embedding technique is that the size of the coefficient forms exactly corresponds to the largest input value. The reason is that value will be the largest exponent in the entire embedding of the corresponding set function.

In the following, we instantiate the framework for C_{out} and $C_{\max}$, respectively. In §5, we show a simpler algorithm to optimize for $C_{\max}$ that bypasses the need for the embedding technique.

4.3 Instantiating C_{out}

In the case of C_{out}, we are working in the $(\min, +)$ semiring. To implement the embedding, we simply need to specify how the "+" operator should work – in the most general form, the "$\otimes$" operator; compare Eq. (2). This corresponds to polynomial multiplication in the coefficient form. Let P_1 and P_2 be two polynomials in coefficient form. Then $P_1 \otimes P_2$ for an exponent e is defined as

$$(P_1 \otimes P_2)(e) = \sum_{\substack{(e_1,c_1)\in P_1 \\ (e_2,c_2)\in P_2 \\ e_1+e_2=e}} c_1c_2. \tag{11}$$

Since the maximum value of the C_{out} cost function could be Wn (recall that W is the largest join cardinality) and assuming a FFT-based implementation of the convolution in Eq. (11), the factor τ_{out} for supporting C_{out} is $O(Wn \log Wn)$.

4.4 Instantiating $C_{\max}$

We now specify the embedding for the $(\min, \max)$ semiring. Unlike C_{out}, we need to specify how the "max" operator should work. Namely, the coefficient of exponent e of two polynomials P_1 and P_2 in coefficient form reads:

$$(P_1 \otimes P_2)(e) = \sum_{\substack{(e_1,c_1)\in P_1 \\ (e_2,c_2)\in P_2 \\ \max(e_1,e_2)=e}} c_1c_2. \tag{12}$$

The intuition is that all exponents *below* e contribute to its final coefficient. If used as in Eq. (12), the size of the coefficient form will still be W, as in the case of C_{out}; this is prohibitively expensive. While there is indeed a way to mitigate this and obtain a running time of $O(2^n n^4)$, which is independent of W, we discovered a much simpler algorithm with an even better running time of $O(2^n n^3)$, which does not require the embedding technique. This is understandable due to the fact that, in the $(\min, \max)$ semi-ring, we are not creating *new* values, as is the case in C_{out}.

To not burden the reader with the technicalities of this approach, we will present directly the simpler algorithm as part of the next section. Apart from being simpler, the algorithm we will present is also practical, as we will show as part of §6.

Algorithm 3: Simpler `DPconv[max]`: Optimal cost w.r.t. $C_{\max}$ in $O(2^n n^3)$-time

1: **Input:** Query graph $Q = (V, E)$
2: **Output:** Optimal cost value w.r.t. $C_{\max}$
3: $cs \leftarrow$ `sort`$([c(S) \mid S \subseteq [|V|]],$ `decreasing=True`$)$
4: $p, step \leftarrow 0, 2^{|V|-1}$
5: **while** $step > 0$ **do**
6: $\quad \gamma \leftarrow cs[p + step]$
7: $\quad \text{DP} \leftarrow \text{LayeredDP}([c \leq \gamma])$
8: $\quad$**if** $\text{DP}(V) > 0$ **then**
9: $\quad\quad p \leftarrow p + step$
10: $\quad$**end if**
11: $\quad step \leftarrow step \,/\, 2$
12: **end while**
13: **return** $cs[p]$

5. A SIMPLE ALGORITHM FOR $C_{\max}$

While our framework can support $C_{\max}$ (see §4.4), we found a much simpler algorithm that does *not* require an intricate implementation.

Key Idea. The key insight is the following: Since we are applying only "min" and "max" operations, the optimal solution will take its value in the set of join cardinalities. Hence, we can *binary search* the optimal value OPT. To check whether a given value γ qualifies to be an optimal solution, we apply a technique used by Kosaraju for exact $(\min, \max)$ *sequence* convolution [20], which we will use on the DP-table itself. The strategy is to first put the DP-entries l.e.q. γ on 1 and those greater than γ on 0, and then run FSC, in the $(+, \times)$ ring, on this modified DP-table. In particular, this refers to one of the layers of the DP-table. We also use our improved layered dynamic programming, described in detail in Ref. [40, Sec. 5], which is a highly tuned implementation of the dynamic program for the $(+, \times)$ ring; for conciseness, we skip its technical details here.

Pseudocode. We outline the pseudocode of the algorithm in Alg. 3. It first *sorts* the join cardinalities in descending order and then performs a binary search on them, searching for the one which separates feasible γ's from infeasible ones (note that the maximum join cardinality is always feasible, but may not be the optimum). To this end, we employ Iverson's bracket notation: Given a property P, $[P]$ returns 1 is the property is true, 0 otherwise. In our case, $[c \leq \gamma]$ is the following:

$$S \mapsto \begin{cases} 1, & \text{if } c(S) \leq \gamma, \\ 0, & \text{otherwise.} \end{cases}$$

Once the DP-table has been computed, the algorithm checks whether this value was feasible, i.e., whether V has a positive value in the DP-table. If that is the case, we search for smaller γ's. The algorithm concludes by returning the smallest γ for which $\text{DP}(V)$ is still positive. In the same manner as for the standard `DPconv`, we can build the join tree once we found the optimal value (see Alg. 2).

Running Time. Via LayeredDP [40, Sec. 5], our new Alg. 3 runs in time $O(2^n \log 2^n + 2^n n^2 \log 2^n) = O(2^n n^3)$. The additional factor $\log 2^n$ in the second term comes from the running time of the binary search on the 2^n-sized sorted list of join cardinalities.

6. EVALUATION

We show by means of experiments that DPconv achieves a significant speedup over the standard $O(3^n)$-time join ordering algorithm.

Experimental Setup. We perform our experiments on a EC2 instance (c5.xlarge) which has an Intel Xeon Platinum 8275CL processor with 4 vCPUs and 8 GB of memory. All join ordering algorithms are implemented in C++.

Benchmark Sets. In the context of this shortened paper, we benchmark the algorithms on clique queries, for which we generate random join cardinalities $\leq$ 100M, with the constraint that $c(S) \leq c(S_1)c(S_2), \forall S_1, S_2, \subsetneq S, S_1 \cap S_2 = \varnothing, S_1 \cup S_2 = S$, i.e., we do not exceed the cardinality of the cross-product of any possible combination of subset pairs. Note that since we directly optimize on clique queries, the running times can also be considered as that of optimizing for cross-products, as discussed in §4.1. Moreover, since subset convolution does not (yet) exploit sparsity—in our case, corresponding to unconnected query subgraphs—, the running time is independent of the cyclicity of the query graph (see §8 for a discussion on possible extensions).

Competitors. The standard exact algorithms, DPccp and DPsub, follow the implementation in the reproducibility experiment of Neumann and Radke [29].[6] We also implement the bitsets as 64-bit integers, which we wrap with helper functions to provide iterators of subsets. The optimization time includes the time for extracting the join tree from the layered dynamic programming (see Alg. 2).

6.1 Super-Polynomial Speedup

Within our framework, DPconv, we have shown that join ordering can be done faster than $O(3^n)$. Specifically, we provided an $O(2^n n^2 W n \log W n)$-time algorithm for optimizing C_{out}, which is $\widetilde{O}(2^n)$ when the largest join cardinality W is polynomial in n, and an $O(2^n n^3)$-time algorithm for optimizing $C_{\max}$; this is the first super-polynomial speedup for the join ordering problem. While the algorithm for C_{out} is not a practical one, we devised in §5 a simple and, *at the same time*, practical algorithm for $C_{\max}$.

DPconv **vs.** DPsub**.** We benchmark on clique queries, as these are the hardest queries to optimize for [28]. In particular, DPsub excels at this type of queries since DPccp has the overhead of exploring the graph itself (note that this is also the case in the experiments of the original paper [26]). We show the optimization times for cliques of up to 24 relations in Fig. 4. The optimization time is averaged for each $n \in \{3, \ldots, 24\}$ across 5 randomly generated instances.

The first observation is that our new algorithm is indeed practical: It starts being faster than DPsub after 17 relations, and for a large join query of 24 relations, it has a speedup of 29x. Note that $n = 17$ is still in the regime of the JOB benchmark. However, JOB has sparse query graphs, hence DPccp is enough for such queries.

7. RELATED WORK

The literature on join ordering is extensive. This is partly because of the effect that a bad join order can have on the query performance and hence the natural desire to avoid such cases. As a result, there are a few *exact* algorithms, a small number of *polynomial-time* algorithms for restrictive cases, several *greedy* (non-optimal) algorithms, and a handful of optimizers based on general-purpose solvers. Our work falls into the category of exact algorithms. In particular, no previous work has observed the link to subset convolution, neither did it achieve a running time as we propose. We are the first to break the $O(3^n)$ time-barrier for the join ordering problem on generic query graphs (and bushy solutions). We divide the related work into exact, approximation, and best-effort algorithms. The latter are either polynomial-time algorithms for special instances or greedy algorithms without any approximation guarantee.

[6] https://db.in.tum.de/~radke/papers/hugejoins-reproducibility.pdf

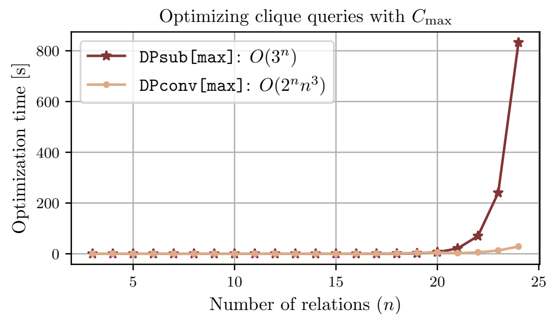

Figure 4: Clique queries optimization: Both DPsub[max] and DPconv[max] optimize for $C_{\max}$.

7.1 Exact Algorithms

The history of the join ordering problem starts at Selinger, proposing an $O(4^n)$-time algorithm, commonly referred to as DPsize [38]. To some extent, this algorithm *does* subset convolution in the naive way, i.e., it iterates all subsets T of a given set S of relations, but does not do that in time $2^{|S|}$, but rather in time 2^n. Vance and Maier [43] observed this limitation and fixed it within the DPsub algorithm, which takes time $O(3^n)$. Since $O(3^n)$ seemed rather rigid, not being adaptive to the graph topology, Ono and Lohman [31] analyzed the *minimum* number of subplan pairs that have to be iterated in any dynamic program. To this end, Moerkotte and Neumann [26] designed DPccp, which emulates to the graph topology and obtains as time-bound exactly the number of connected complement pairs (#ccp's). However, the running time $O(3^n)$ still persisted. In their recent work, Haffner and Dittrich [15] showed that using the A* algorithm, one can obtain an algorithm which still outputs the optimal plan without having to explore all #ccp's. This is indeed a promising result, as it shows that the lower-bound of #ccp can in some cases be by-passed. However, in the worst case, the running time is still the unyielding $O(3^n)$. In our work, we obtain for the first time an $O^*(2^n W)$-time algorithm, completely breaking the $O(3^n)$ time-barrier when W is polynomial in n. In the case of $C_{\max}$, i.e., minimizing the maximum intermediate join cardinality, we obtain an $O(2^n n^3)$-time algorithm, which is also practical.

Bottom-Up vs. Top-Down. It is well known that dynamic programs have two implementations, *bottom-up* and *top-down*, each with its advantages and disadvantages. One of the most compelling advantages of top-down enumeration

is the possibility of easily integrating cost-bounds so that the search space may be easily pruned [13]. Hence, Chaudhuri et al. [5] explore the possiblity of implementing join ordering as a top-down procedure, only considering linear solutions. Building on this work, DeHaan and Tompa [8] extend the top-down method to bushy join trees, disallowing cross products. Fender and Moerkotte [11, 12] improve the running time of these algorithms and get rid of the connectedness check, i.e., only outputting the ccp's.

7.2 Approximation Algorithms

Exact algorithms are rather expensive. To this end, Chatterji et al. [4] analyzed whether there are instances that can be solved by approximation algorithms in polynomial time. Unless $\mathsf{P} = \mathsf{NP}$, the answer remains negative. Specifically, they showed that, for any $\delta > 0$, the problem of approximating the optimal cost K within a factor of $2^{\Theta(\log^{1-\delta} K)}$ is NP-hard. (Note that our $(1+\varepsilon)$-approximation algorithm in Ref. [40, Sec. 7] still runs in *exponential* time.)

7.3 Best-Effort Algorithms

The NP-hardness of a fundamental problem is a bitter truth. Hence, research has focused on finding polynomial-time algorithms for special instances or greedy algorithms for arbitrary query graphs.

Polynomial-Time Algorithms. Exponential-time algorithms fail to optimize larger queries in a reasonable time. To this end, it is interesting to ask which instances admit *polynomial-time* algorithms. The most notable one is the cubic-time algorithm for chain queries. Another class is that of tree queries, for which the IKKBZ algorithms returns the optimal left-deep join tree [18, 22]. Neumann and Radke [29] observed that one can use IKKBZ as a sub-routine: They *linearize* the query graph via IKKBZ (since a left-deep solution is inherently a linear ordering of the underlying graph) and then run the cubic-time dynamic program on top to build a near-optimal solution. This strategy yields excellent costs for tree queries.

Greedy Algorithms. Research has also focused on greedy algorithms which can at least *avoid* the bad plans. The most representative is the Greedy Operator Ordering (GOO) [10] that chooses the cheapest sub-plan at each step. This runs in $O(n \log n)$-time, yet it does not come with any optimality guarantee on the output join order. This gap between exponential-time exact algorithms and purely greedy ones has remained unexplored until Kossman and Stocker [21] introduced Iterative Dynamic Programming (IDP) which refines the greedy join orders of large queries. The key insight is to iteratively run exact DP on join subtrees of size k.

General-Purpose Solvers. Join ordering has also been approached by several general-purpose solvers, such as genetic algorithms [39], mixed-integer linear programming [41], and simulated annealing [39]. Note that these works only approximate the optimal solution (*without* any approximation guarantee). The problem can also be optimized on quantum hardware via quantum annealing [45, 36]. However, this does not lower the *classical* time-complexity of exact join ordering. Motivated by the promise of workload-aware query optimization, research also has focused on *learned* alternatives: Marcus and Papaemmanouil [25] suggest using Reinforcement Learning and introduce an agent that outputs the join order and is penalized based on the corresponding join cost. Motivated by the repetitiveness of the queries in cloud workloads [35], a further promising direction is query super-optimization [24].

8. DISCUSSION

Resource-Aware Query Optimization. The trend nowadays is to execute queries in multi-tenant cloud machines. Recently, Viswanathan et al. [44] made the case for *resource-aware* query optimization. The $C_{\max}$ cost function can serve as a proxy for the maximum memory consumption of a given query. Minimizing $C_{\max}$ of concurrently running queries can help reduce memory spikes.

Co-Optimizing C_{out} and $C_{\max}$. The cloud data warehouse Amazon Redshift uses predicted query memory to make scheduling decisions [34]. Instead, one could follow a proactive approach in which a query's runtime and memory consumption is co-optimized with query scheduling. For example, when there is a high (concurrent) memory load on the system, one would minimize the peak memory consumption of newly arriving queries, while when there is low memory load, one can afford a higher memory consumption. Likewise, if there are long-running queries with a low memory footprint in the system, one might opt for a query plan that uses more memory in exchange for faster execution.

More Practical Implementations. While we break the $O(3^n)$ time-barrier in the theoretical sense and, indeed, also provide a practical implementation for $C_{\max}$ running in $O(2^n n^3)$-time, it is interesting to further explore practical implementations for C_{out}, both for the exact (§4.3) and the approximation algorithm [40, Sec. 7].

Sparse Subset Convolution. Subset convolution does not (yet) have a *sparse* counterpart, as is the case for sequence convolution (we refer the reader to Jin and Xu [19] for the latest results on sparse sequence convolution). This would be particularly useful for sparse query graphs.

9. CONCLUSION

Join ordering, or finding the optimal order of the joins of a query, is an indispensable task in a database management system. The problem has its roots in the seminal work of Selinger et al. [38], culminating with the graph-theoretic exact algorithm by Moerkotte and Neumann [26]. Despite recent research by Haffner and Dittrich [15], the worst-case running time has remained $O(3^n)$.

In this work, we showed the first super-polynomial speedup over the standard dynamic programming solution. Our framework optimizes (i) C_{out} in $O^*(2^n)$-time, when the largest join cardinality W is polynomial in n, and (ii) $C_{\max}$ in $O(2^n n^3)$-time. DPconv is based on subset convolution, a fundamental tool in parameterized algorithms [7], and uses the fact that join ordering is implicitly a dynamic programming recursion using subset convolution similar to other classic problems in the algorithm design literature (see Björklund et al. [2]).

Beyond the theoretical results, we have made DPconv practical for database systems. In particular, our algorithm for optimizing $C_{\max}$ outperforms the standard exact algorithm for cliques with 17 relations and more.

We expect future work on sparse subset convolution to further speed up our framework for query graphs with few connected subgraphs.

10. REFERENCES

[1] R. E. Bellman. *The Theory of Dynamic Programming.* RAND Corporation, Santa Monica, CA, 1954.

[2] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets möbius: fast subset convolution. In D. S. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 67–74. ACM, 2007.

[3] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.

[4] S. Chatterji, S. S. K. Evani, S. Ganguly, and M. D. Yemmanuru. On the complexity of approximate query optimization. In L. Popa, S. Abiteboul, and P. G. Kolaitis, editors, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 282–292. ACM, 2002.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 190–200. IEEE Computer Society, 1995.

[6] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[7] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Algebraic techniques: sieves, convolutions, and polynomials*, pages 321–355. Springer International Publishing, Cham, 2015.

[8] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 785–796. ACM, 2007.

[9] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[10] L. Fegaras. A new heuristic for optimizing large queries. In G. Quirchmayr, E. Schweighofer, and T. J. M. Bench-Capon, editors, *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer, 1998.

[11] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 864–875. IEEE Computer Society, 2011.

[12] P. Fender and G. Moerkotte. Reassessing top-down join enumeration. *IEEE Trans. Knowl. Data Eng.*, 24(10):1803–1818, 2012.

[13] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 414–425. IEEE Computer Society, 2012.

[14] F. V. Fomin and D. Kratsch. *Exact Exponential Algorithms.* Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[15] I. Haffner and J. Dittrich. Efficiently computing join orders with heuristic search. *Proc. ACM Manag. Data*, 1(1):73:1–73:26, 2023.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968.

[17] D. Havenstein, P. Lysakovski, N. May, G. Moerkotte, and G. Steidl. Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs. In A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, editors, *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 546–554. OpenProceedings.org, 2020.

[18] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[19] C. Jin and Y. Xu. Shaving logs via large sieve inequality: Faster algorithms for sparse convolution and more. In B. Mohar, I. Shinkar, and R. O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1573–1584. ACM, 2024.

[20] S. Kosaraju. Efficient tree pattern matching. In *30th Annual Symposium on Foundations of Computer Science*, pages 178–183, 1989.

[21] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[22] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann, 1986.

[23] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In H. Boral and P. Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 18–27. ACM Press, 1988.

[24] R. Marcus. Learned query superoptimization. In R. Bordawekar, C. Cappiello, V. Efthymiou, L. Ehrlinger, V. Gadepally, S. Galhotra, S. Geisler, S. Groppe, L. Gruenwald, A. Y. Halevy, H. Harmouch, O. Hassanzadeh, I. F. Ilyas, E. Jiménez-Ruiz, S. Krishnan, T. Lahiri, G. Li, J. Lu, W. Mauerer, U. F. Minhas, F. Naumann, M. T. Özsu, E. K. Rezig, K. Srinivas, M. Stonebraker, S. R. Valluri, M. Vidal, H. Wang, J. Wang, Y. Wu, X. Xue, M. Zaït, and K. Zeng, editors, *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data*

Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023, volume 3462 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.

[25] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

[26] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 930–941. ACM, 2006.

[27] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In J. T. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552. ACM, 2008.

[28] T. Neumann. Query simplification: graceful degradation for join-order optimization. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 403–414. ACM, 2009.

[29] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 677–692. ACM, 2018.

[30] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.

[31] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.

[32] O. Ponta, F. Hüffner, and R. Niedermeier. Speeding up dynamic programming for some np-hard graph recoloring problems. In M. Agrawal, D. Du, Z. Duan, and A. Li, editors, *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, volume 4978 of *Lecture Notes in Computer Science*, pages 490–501. Springer, 2008.

[33] B. Radke and T. Neumann. Lindp++: Generalizing linearized DP to crossproducts and non-inner joins. In T. Grust, F. Naumann, A. Böhm, W. Lehner, T. Härder, E. Rahm, A. Heuer, M. Klettke, and H. Meyer, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, volume P-289 of *LNI*, pages 57–76. Gesellschaft für Informatik, Bonn, 2019.

[34] G. Saxena, M. Rahman, N. Chainani, C. Lin, G. Caragea, F. Chowdhury, R. Marcus, T. Kraska, I. Pandis, and B. M. Narayanaswamy. Auto-wlm: Machine learning enhanced workload management in amazon redshift. In S. Das, I. Pandis, K. S. Candan, and S. Amer-Yahia, editors, *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 225–237. ACM, 2023.

[35] T. Schmidt, A. Kipf, D. Horn, G. Saxena, and T. Kraska. Predicate caching: Query-driven secondary indexing for cloud data warehouses. In P. Barceló, N. Sánchez-Pi, A. Meliou, and S. Sudarshan, editors, *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, pages 347–359. ACM, 2024.

[36] M. Schönberger, S. Scherzinger, and W. Mauerer. Ready to leap (by co-design)? join order optimisation on quantum hardware. *Proc. ACM Manag. Data*, 1(1):92:1–92:27, 2023.

[37] J. Scott, T. Ideker, R. M. Karp, and R. Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. In S. Miyano, J. P. Mesirov, S. Kasif, S. Istrail, P. A. Pevzner, and M. S. Waterman, editors, *Research in Computational Molecular Biology, 9th Annual International Conference, RECOMB 2005, Cambridge, MA, USA, May 14-18, 2005, Proceedings*, volume 3500 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2005.

[38] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In P. A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34. ACM, 1979.

[39] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997.

[40] M. Stoian and A. Kipf. DPconv: Super-Polynomially Faster Join Ordering. *Proc. ACM Manag. Data*, 2(6):234:1–234:26, 2024.

[41] I. Trummer and C. Koch. Solving the join ordering problem via mixed integer linear programming. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1025–1040. ACM, 2017.

[42] B. Vance. *Join-order Optimization with Cartesian Products*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.

[43] B. Vance and D. Maier. Rapid bushy join-order

optimization with cartesian products. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 35–46. ACM Press, 1996.

[44] L. Viswanathan, A. Jindal, and K. Karanasos. Query and resource optimization: Bridging the gap. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1384–1387. IEEE Computer Society, 2018.

[45] T. Winker, U. Çalikyilmaz, L. Gruenwald, and S. Groppe. Quantum machine learning for join order optimization using variational quantum circuits. In S. Groppe, L. Gruenwald, and C. Hsu, editors, *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE 2023, Seattle, WA, USA, 18 June 2023*, pages 5:1–5:7. ACM, 2023.

[46] F. Yates. The design and analysis of factorial experiments. *Imperial Bureau of Soil Science*, 1937.