DPconv: Super-Polynomially Faster Join Ordering

MIHAIL STOIAN, UTN, Germany
ANDREAS KIPF, UTN, Germany

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.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: join ordering, dynamic programming, fast subset convolution, exponential-time approximation algorithm

ACM Reference Format:
Mihail Stoian and Andreas Kipf. 2024. DPconv: Super-Polynomially Faster Join Ordering. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 234 (December 2024), 26 pages. https://doi.org/10.1145/3698809

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 [20]. 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 introduces the first dynamic program to (exactly) optimize the ordering problem [44]. 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 [48, 49].

Later, Moerkotte and Neumann [30] 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 hypergraphs [31]). 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 [36]. Recently, Haffner and Dittrich [18] 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,

Authors' addresses: Mihail Stoian, UTN, Ulmenstraße 52i, Nuremberg, Germany, 90443, mihail.stoian@utn.de; Andreas Kipf, UTN, Ulmenstraße 52i, Nuremberg, Germany, 90443, andreas.kipf@utn.de.

ACM 2836-6573/2024/12-ART234
https://doi.org/10.1145/3698809

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

$$\overset{V := \boxed{R_1R_2R_3R_4}}{=} \underset{\textbf{Subset Convolution}}{\boxed{\min_{\emptyset\subset S\subset V} \left(\mathrm{DP}[S] + \mathrm{DP}[V\setminus S]\right)}}$$

Fig. 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* [2].

as known from the A^* algorithm [19]. 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 [48, 49] and DPccp [30] 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 of the naive evaluation. To this end, we develop a new exact algorithmic framework based on fast subset convolution that has super-polynomial speedup over DPccp / DPsub. This breaks 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 $\widetilde{O}(2^n)$ 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.

We further reduce the optimization time for C_{out} to $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time using an $(1+\varepsilon)$-approximation algorithm. Unlike our exact algorithm, the running time of this algorithm is independent of W.

In addition, we devise a new cost function which combines the benefits of C_{out} and $C_{\max}$ and provide an implementation which first computes the optimal $C_{\max}$ value and then runs a pruned C_{out} optimization. We show that this optimization is faster than that of the "vanilla" C_{out} when using our new framework.

[1]The notation $\widetilde{O}$ hides poly-logarithmic factors; in this particular case, $n^{O(1)}$.

Contribution. We summarize our contributions below:

(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.
(3) We introduce an $(1+\varepsilon)$-approximation algorithm for the join ordering problem under C_{out} in $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time.
(4) We initiate the joint study of C_{out} and $C_{\max}$: Minimize the sum of the intermediate join sizes so that the largest one is equal to the optimal $C_{\max}$ value.

Running Times. Let us first relate the running times for C_{out} and $C_{\max}$, which seem quite disparate at first glance. They both rely on our highly-tuned implementation of fast subset convolution for dynamic programming (Sec. 5), which runs in $O(2^n n^2)$-time. Thus, we can observe a common $O(2^n n^2)$-time factor in both running times. The difference lies in the implementation of the individual cost functions: Optimizing for C_{out} introduces an additional $O(Wn \log Wn)$-time factor resulting from the application of FFT to sequences of length Wn (Sec. 3.3), while $C_{\max}$ incurs only an $O(n)$-time overhead (Sec. 6).

Search Space. DPconv imposes no restrictions on the shape of the query graph or join tree. Our framework optimizes arbitrary query graphs—both acyclic and cyclic queries—including cliques, which are considered the worst case of join ordering [33], *and* bushy join trees, as do other join ordering algorithms such as DPsub and DPccp. This includes optimizing for cross-products in the same running time as for arbitrary query graphs. We discuss this aspect as part of Sec. 2. Note that our framework also optimizes query hypergraphs, representing non-inner joins [31] (we discuss this aspect in Sec. 3.1).

Other Cost Functions. The literature on join ordering also addresses various cost functions beyond C_{out} and $C_{\max}$, depending on how a join is executed, e.g., by sort-merge join, hash-join, or nested-loop join [29]. We demonstrate that our framework can accommodate the cost function associated with the sort-merge join because it satisfies an additive separability property (see Sec. 3.5). However, the cost functions associated with hash-joins and nested-loop joins do not enjoy this property and thus cannot be mapped to our framework.

Organization. The rest of the paper is organized as follows: First, in Sec. 2, we formalize the problem of join ordering and that of fast subset convolution. Then, in Sec. 3, we introduce DPconv along with the novel connection between join ordering and subset convolution. We describe the machinery behind fast subset convolution in Sec. 4, and in Sec. 5 we show how to shave a linear factor from the running time of *any* dynamic programming recursion based on subset convolution (including that of DPconv). Based on this, we provide in Sec. 6 a practical algorithm for $C_{\max}$. Then, we outline the approximation algorithm in Sec. 7. We propose C_{cap} in Sec. 8, which we start for the first time the joint study of C_{out} and $C_{\max}$ with. We outline related work in Sec. 10, provide a discussion in Sec. 11, and finally conclude in Sec. 12.

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 Q is defined as

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

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., $R_i.a = const$, 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 relational algebra [7].

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 [35]. 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 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 represented as a recursive function along a join tree $\mathcal{T}$, as follows:

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

where c is the join cardinality function defined on sets of relations, 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-optimality 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 has been substituted by "+" and "max", respectively. We discuss the applicability of our framework to other cost functions in the literature in Sec. 3.5.

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., $S_1 \cap S_2 = \emptyset$ and $S = S_1 \cup S_2$. Consequently, given a cost function C, the problem can be optimized by the following dynamic programming (DP) recursion, which closely follows the definition of C:

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

Indeed, this is the idea explored by Selinger and the subsequent work [30, 44, 48, 49]. In particular, `DPccp` [30] optimizes 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 [9, 17]. 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 [11] and min-cost k-coloring [9]. 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 use an implicit subset convolution. Through our work, join ordering is now becoming part of this family of problems [3, 9, 11, 37, 43].

While our main result mostly uses FSC in a black-box manner, we present the full machinery behind it in Sec. 4. Note that for an efficient implementation of the dynamic programs, we will revisit the computation of FSC in Sec. 5, and shave a linear factor for generic FSC-based dynamic programs, as well as several constant factors hidden behind the running time's big-O notation.

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 $(+, \cdot)$ ring is defined for *all* $S \subseteq [n] := \{1, \ldots, n\}$ by

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

Let us first make a few observations: First, the above kind of subset convolution, in the $(+, \cdot)$ ring, is not yet what we exactly need in DPconv. In the next paragraphs, we gradually introduce the toolset to support the subset convolution appearing in Eq. (5). Second, naively evaluating the above equation for all subsets S takes $O(3^n)$-time. This follows from the fact that for each S we have to iterate over all its $\binom{n}{|S|}$-many subsets T.[3]

Where The Speedup Comes From. The faster computation in Björklund et al. [2] has its roots in a simple observation: One can do a calculation similar to the FFT algorithm [8]. The reason: FFT was specifically designed to speed up *sequence* convolutions, bringing down the $O(n^2)$-time of the naive algorithm to a (still unbeatable) $O(n \log n)$-time. Its key insight was to (a) map the original sequence into a Fourier space, (b) perform the convolution in that space as a *point-wise* multiplication – which takes linear time instead – and (c) bring the result from the Fourier space back into the original one. The authors take a similar path, resulting in a running time of $O(2^n n^2)$. The perhaps only difference to the original FFT algorithm is *how* the subset functions are mapped to a similar Fourier space where the convolution can be transformed into a point-wise multiplication.

The Real Deal: Semi-Rings. Dynamic programs defined on sets often require the computation to be worked out in *semi-rings*. This is also the case in join ordering for C_{out} and $C_{\max}$: The well-known C_{out} works in the $(\min, +)$ semi-ring, while $C_{\max}$ works instead in the $(\min, \max)$ semi-ring. The $(\min, +)$ subset convolution of two set functions f and g, $h = f \circ g$, is defined for all $S \subseteq [n]$ as

$$h(S) = (f \circ g)(S) = \min_{T \subseteq S} (f(T) + g(S \setminus T)). \tag{6}$$

[3]Formally, $\sum_{k=0}^{n} \binom{n}{k} 2^k = (1+2)^n = 3^n$.

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. $C$
3: $\mathrm{DP}[\varnothing] \leftarrow +\infty$
4: $\mathrm{DP}[\{R_i\}] \leftarrow 0, \forall R_i \in V$
5: for each $k$ in $2, \ldots, |V|$ do
6:     $\mathrm{DP}' \leftarrow \mathrm{FSC}_{(\min,\otimes)}(\mathrm{DP}, \mathrm{DP})$
7:     $\mathrm{DP}[S] \leftarrow \mathrm{DP}'[S] \otimes c(S), \forall S$ s.t. $|S| = k$
8: end for
9: return $\mathrm{DP}[V]$
```

Unlike the previous kind of subset convolution, computing in the $(\min, +)$ semi-ring results in a different time-complexity landscape. Surprisingly enough, in the general setting, the naive $O(3^n)$-time algorithm which we saw before has the best running time so far. However, in the case where the values of the set functions are bounded integers, one can leverage the previous fast subset convolution for the $(+, \cdot)$ ring. We will come to this in the next section.

2.5 Rings & Semi-Rings

We have mentioned rings and semi-rings several times so far. We now want to introduce them in the context of dynamic programming and, more specifically, join ordering. We will focus in particular on the $(+, \cdot)$ ring and the $(\min, +)$ and $(\min, \max)$ semi-rings. Note that we are only aiming for an intuitive understanding of why supporting the latter is much more complex than the simple ring setting, where fast subset convolution can work directly in $O(2^n n^2)$-time, as shown earlier in Sec. 2.4.

Intuition. Within the pair of operators of a (semi-)ring, the first one is decisive. In our case, while the $(+, \cdot)$ ring has "+" as its first operator, both $(\min, +)$ and $(\max, \max)$ have "min". To understand the contrast between these, consider the following illustrative example: If we calculate $2 + 3 + 5 = 10$ and want to remove one of the first terms, e.g., 2, we can recover the sum of the *other* terms by using the inverse of 2, i.e., $10 + (-2) = 8$. Things are not so clear in the case of "min": If we have $\min\{2, 3, 5\} = 2$ and want to remove 2, we cannot simply recover the minimum of the remaining elements, $\min\{3, 5\}$. The underlying problem is that has 2 no inverse. This example may seem artificial at first, but this very problem occurs when applying the inverse map to come back from the Fourier space – essentially step (c) above. How can this be alleviated? What Björklund et al. [2] propose is a standard trick in algorithms: embed the $(\min, +)$ semi-ring in the $(+, \cdot)$ ring. We explain and exemplify this technique in Sec. 3.2.

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.

3 OUR FRAMEWORK

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

3.1 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, Eq. (6). In particular, we show that join

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 $\emptyset \subset T \subset S$ do
5:    if $c(S) \otimes \text{DP}(T) \otimes \text{DP}(S \setminus T) = \text{DP}(S)$ then
6:       return (BuildJoinTree($T$), BuildJoinTree($S \setminus T$))
7:    end if
8: end for
```

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 $\emptyset$ 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 $\text{DP}[\emptyset]$, i.e., the DP-cell representing the empty set of relations, to $+\infty$.[4] Since FSC is called n times, the total optimization time adds up to $O(2^n n^3 \tau_C)$, where the function τ_C is tailored to the specific cost function C and accounts for the time overhead of semi-ring operations. We will cover the exact expressions of τ_C for individual cost functions in the following sections (see Sec. 3.3 for C_{out} and Sec. 3.4 for $C_{\max}$).

Note that we will shave a factor of $O(n)$ from the running time of generic FSC-based dynamic programs, including ours, in Sec. 5, thus reducing the total running time to $O(2^n n^2 \tau_C)$-time for a cost function C.

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 C to which the semi-ring $(\min, \otimes)$ corresponds. It 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 $\text{DP}[V]$.

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 $O(2^n n)$.

[4]This is not mathematically rigorous. One has to *define* what $+\infty$ in the specific semi-ring is.

Cross-Products. While allowing cross-products can lead to better overall costs [36, 38], the search space increases exponentially [36]. A prominent way to deal with cross-products is to heuristically insert them when they are guaranteed to be beneficial [36]; this tends to be the case when the estimated cardinality of the input is small enough [26, 38].

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 [48, 49], 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.

Query Hypergraphs. A standard way to model arbitrary non-inner joins in the query graph is to introduce corresponding binary join hyperedges. A binary join hyperedge $h = (A, B)$ connects two sets of relations A and B [31]. This is a generalization of the regular join edge which connects only two relations. In the query hypergraph setting, whenever we want to join two sets of relations connected by a hyperedge, we have to check that both sides are themselves connected *and* there is a hyperedge connecting them. Fortunately, since Eq. (5) enforces that the join cardinalities are taken into account only *after* the DP-layer has been optimized (lines 6-7, Alg. 1), we can directly specify which subgraphs are connected (using Ref. [31]); this is independent of whether the subgraph contains hyperedges or not. Extending our framework to optimize group-by operators optimally, as in Eich et al. [12], is an interesting future work.

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

3.2 Embedding Technique

Recall the motivating example in the section on rings and semi-rings (Sec. 2.5). To enable the existence of the inverse element, Björklund et al. [2] propose to embed the semi-ring into a ring,

Polynomials to the Rescue. The embedding technique maps the values of the set functions to monomials and then runs the fast subset convolution algorithm in the $(+, \cdot)$ 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 set functions 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 Sec. 6, we show a simpler algorithm to optimize for $C_{\max}$ that bypasses the need for the embedding technique.

3.3 Instantiating C_{out}

In the case of C_{out}, we are working in the $(\min, +)$ semi-ring. 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

$$(P_1 \otimes P_2)(e) = \sum_{\substack{(e_1,c_1) \in P_1 \\ (e_2,c_2) \in P_2 \\ e_1+e_2=e}} c_1 c_2. \tag{7}$$

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. (7), the factor τ_{out} for supporting C_{out} is $O(Wn \log Wn)$.

3.4 Instantiating $C_{\max}$

We now specify the embedding for the (min, max) semi-ring. 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:

$$(P_1 \otimes P_2)(e) = \sum_{\substack{(e_1,c_1) \in P_1 \\ (e_2,c_2) \in P_2 \\ \max(e_1,e_2)=e}} c_1 c_2. \tag{8}$$

The intuition is that all exponents *below* e contribute to its final coefficient. If used as in Eq. (8), 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 the first approach, we will present directly the simpler algorithm; we continue its presentation in Sec. 6.

3.5 Beyond C_{out} and $C_{\max}$

Beside C_{out} and $C_{\max}$, literature on join ordering also considers other cost functions: Moerkotte [29] mentions cost functions related to (a) nested-loop, (b) hash, and (c) sort-merge joins. These cost functions have been mainly designed for left-deep join trees. However, it is an easy exercise to remodel them to work on bushy joins trees. We show that DPconv can be extended to the cost function associated to the sort-merge join.

The problem is that these cost functions require that $c(T)$ be rewritten in terms of $\mathcal{T}_1$ and $\mathcal{T}_2$. That is, instead of $c(T)$, we would need to write $c(T_1, T_2)$; see the below example. The key idea to solve this is to first check whether $c(T_1, T_2)$ can be separated into two independent terms depending only on T_1 and T_2, respectively.

When It Works. We take as running example the sort-merge join cost. Adapting the definition by Moerkotte [29, Sec. 3.1.3], we have:

$$C_{\text{smj}}(\mathcal{T}) = \begin{cases} 0, & \text{if } \mathcal{T} \text{ is a single relation} \\ c(T_1)\log c(T_1) & \\ \quad + c(T_2)\log c(T_2) & \\ \quad + C_{\text{smj}}(\mathcal{T}_1) + C_{\text{smj}}(\mathcal{T}_2), & \text{if } \mathcal{T} = \mathcal{T}_1 \bowtie \mathcal{T}_2, \end{cases} \tag{9}$$

where T_1 and T_2 are the set of relations corresponding to $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $c(T_1)$ and $c(T_2)$ are the corresponding join cardinalities. Note the change to our original $c(T)$ in Eq. (2): The split $\mathcal{T} = \mathcal{T}_1 \bowtie \mathcal{T}_2$ now plays a role. For our framework, this means that we can no longer simply optimize the subset convolution part separately (see line 6 of Alg. 1). We also need to account for the actual sort-merge join cost at the current join, $c(T_1)\log c(T_1) + c(T_2)\log c(T_2)$. However, there is a simple solution to fix this: We can separate the sort-merge join cost and integrate that into the corresponding side, i.e., either $\mathcal{T}_1$ or $\mathcal{T}_2$. Concretely, we need to modify line 6 in Alg. 1 as follows:

$$\mathrm{FSC}_{(\min,+)}(\mathrm{DP} + c \log c),$$

where the inner function, DP + $c \log c$, is applied point-wise to each set $S \subseteq [n]$. Put simple, we also add to each DP-entry the sort-merge join cost corresponding to each side. This does not incur a large overhead in the optimization time, as the addition can be performed when pre-processing the zeta transforms of the DP-layers (see Sec. 5.1).

When It Does Not Work. Note that this adaptation to the sort-merge join worked because we could split the initial $c(T_1, T_2)$ into two separate cost factors that could be "sinked" in the entries of the DP table. To support other cost functions, they need to have a similar *additive* separation property. For instance, the nested-loop join cost, $c(T_1)c(T_2)$, does *not* enjoy this property. This means that our subset convolution based framework, `DPconv`, cannot be extended to this cost function. A similar situation holds for the hash-join cost, at least in the (classic) setting we are considering: $c(T_1, T_2) = 1.2 \max\{c(T_1), c(T_2)\}$. This extension from Moerkotte's definition mainly designed for left-deep join trees [29, Sec. 3.1.3] takes into account that the hash-table is built on the smaller side, known as the build side. The issue is that "max" in $c(T_1, T_2)$ destroys its additive separability into two cost functions depending only on T_1 and T_2. Therefore, the hash-join cost function cannot also be supported in our framework.

4 FAST SUBSET CONVOLUTION

We next describe Fast Subset Convolution (FSC). We take a closer look at FSC from a practical perspective, so that in Sec. 5 we can shave the promised $O(n)$-time factor from the running time of `DPconv`. In the following, we adopt the notation from the Parameterized Algorithms book [9], since it has established itself in the literature compared to that of Björklund et al. [2].

4.1 Zeta Transform

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

$$(\zeta f)(S) = \sum_{T \subseteq S} f(T), \tag{10}$$

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 Sec. 4.4.

4.2 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,

$$(\zeta h)(S, r) = \sum_{d=0}^{r} (\zeta f)(S, d)(\zeta g)(S, r - d), \tag{11}$$

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 $d \leq r$.

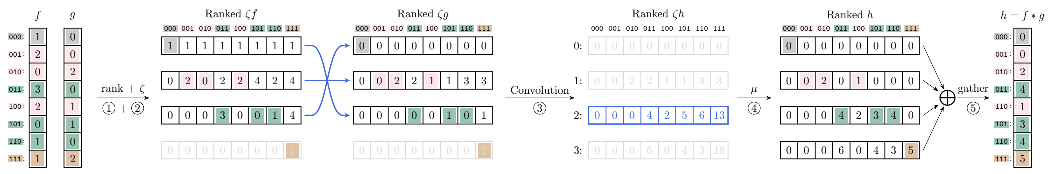

Fig. 2. Visualizing the fast subset convolution (FSC), outlined in Lst. 3: ① We rank the set functions f and g and ② apply the zeta transform to obtain ζf and ζg, respectively. ③ We perform the ranked convolution between ζf and ζg. ④ We apply the Möbius transform to obtain the ranked h. ⑤ Finally, we reconstitute $h = f * g$, the actual subset convolution. We highlight in color the steps needed to compute the second rank "slice" of ζh, namely $(\zeta h)(:, 2)$, during ranked convolution (as in Sec. 4.2). 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 $(\zeta g)(:, 0)$.

4.3 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

$$(\mu f)(S) = \sum_{T \subseteq S} (-1)^{|T|} f(T). \tag{12}$$

A full-fledged example of FSC is shown in Sec. 4.5 and its associated Fig. 2.

4.4 Implementation

4.4.1 Zeta Transform. A naive evaluation of Eq. (10) 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 [52]. Define $\hat{f}_0(S) = f(S)$ for all $S \subseteq [n]$, and then iterate for all $j = 1, 2, \dots, n$ and $S \subseteq [n]$ as follows [2]:

$$\hat{f}_j(S) = \begin{cases} \hat{f}_{j-1}(S) & \text{if } j \notin S, \\ \hat{f}_{j-1}(S \setminus \{j\}) + \hat{f}_{j-1}(S) & \text{if } j \in S. \end{cases} \tag{13}$$

By induction, one can show that $\hat{f}_n(S) = (\zeta f)(S)$ for all $S \subseteq [n]$. The computation of Eq. (13) 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. (13).

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

Listing 1. Zeta transform

4.4.2 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]:

$$\check{f}_j(S) = \begin{cases} \check{f}_{j-1}(S) & \text{if } j \notin S, \\ -\check{f}_{j-1}(S \setminus \{j\}) + \check{f}_{j-1}(S) & \text{if } j \in S. \end{cases} \tag{14}$$

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.

```
FSC(f, g):
  for (r in [0, n]):
    // Rank f and g
    for (S with |S| = r):                              ①
      f[S, r] = f[S]
      g[S, r] = g[S]

    // Zeta transform                                  ②
    zf[S, r] = zeta(f[:, r])
    zg[S, r] = zeta(g[:, r])

    // Ranked convolution                              ③
    for (d in [0, r])
      zh[:, r] += zf[:, d] * zg[:, r - d]
                                                       ④
    // Moebius transform
    h[:, r] = mu(zh[:, r])
                                                       ⑤
  // Reconstitute h
  for (S in [0, 2**n]):
    h[S] = h[S, |S|]
```

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

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.

4.5 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. (10): 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 $(\zeta g)(111, 1)$ is made up of the non-zero values $g(010)$ and $g(100)$.

③ **Ranked Convolution.** Once both ranked ζf and ζg have been computed, we can run the (ranked) convolution between them, as described in Eq. 11. 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. (11), we simply perform a dot product between these and sum up the results. For instance, to obtain $(\zeta h)(111, 2)$, 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 Sec. 4.3, 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. (12), 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, $h(111, 2)$ is computed as follows: The values $\zeta h(\texttt{100}, 2), \zeta h(111, 2)$ are at odd cardinalities, so we subtract their values, while $\zeta h(\texttt{100}, 2), \zeta h(\texttt{101}, 2)$, and $\zeta h(\texttt{110}, 2)$ are at even cardinalities, so we add them. In total, these results in $-4 - 13 + 2 + 5 + 6 = 0$, which is exactly $h(111, 2)$.

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.

4.6 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.

When used *as is* in dynamic programming recursions, the running time of FSC is multiplied by a factor $O(n)$. To this end, we show in the next section how to improve the running time from $O(2^n n^3)$ to $O(2^n n^2)$. To motivate this, note that a slowdown of 20x for $n = 20$ in the context of join ordering can make the difference between a practical and an impractical algorithm.

5 LAYERED DYNAMIC PROGRAMMING

Subset convolution is usually employed in definitions of dynamic programs (as our own) where it is called to optimize the kth layer of the DP-table (in our case, line 6 in Alg. 1). As previously argued, this call contains a lot of redundancy.

A first observation, ($\star$), is that, even though we call FSC on the *entire* DP-table (Alg. 1, lines 6-7), we will only update the table for subsets S of size exactly k. Taking a look at the internals of FSC explained in Sec. 4, we observe that the ranked convolution, Eq. (11), actually computes $\hat{h}(:, r)$ for *each* r in *each* call. This is detrimental, as we will use only the kth layer $\hat{h}(:, k)$ in the kth call. A second observation, ($\star\star$), is that the DP-table itself does not change for subsets of size less than k.

In the following, we explain how one can improve the computation of the transforms and that of the ranked convolution given these two observations to reduce the time-complexity of FSC-based DPs from $O(2^n n^3)$ to $O(2^n n^2)$, hence shaving a $O(n)$ factor.

5.1 Layer-Wise Zeta Transform

With this setting in mind, we can adapt the zeta transforms[5] intrinsically used in Alg. 1 to run faster. Namely, at layer $k > 1$, we do not need to *recompute* the zeta transforms $(\zeta f)(:, j)$, with $j < k$, since due to observation ($\star\star$), these do *not* change once computed and can, hence, be cached and reused during the entire computation. Consequently, at the kth call to FSC, we only need to compute $(\zeta f)(:, k - 1)$.

[5]The plural is intended.

5.2 Layer-Wise Ranked Convolution

In the same manner, by observation (★), we may skip the outer for-loop (line 2, Lst. 3) and directly compute $(\zeta h)(:, k)$ once $(\zeta f)(:, k - 1)$ has been computed as previously argued. Notably, due to symmetry – recall that we actually call FSC with the DP-table – we may only iterate d until $\lfloor \frac{i}{2} \rfloor$ and multiply $f(:, d)g(:, i-d)$ by 2 (apart from the case when d is indeed equal to $i-d$). Finally, we apply the Möbius transform on $(\zeta h)(:, k)$ to obtain the kth layer of the DP-table (this is symbolically denoted by DP′ in Alg. 1).

Alone these two optimizations shave an $O(n)$-overhead from the running time. We, however, present an additional optimization which, albeit does not reduce the asymptotic time complexity, it does indeed save another constant factor.

5.3 Avoiding Useless Multiplications

Recall that our algorithms will work with coefficient forms of polynomials, as described in Sec. 3.2. Hence, the multiplication operator in the ranked convolution (Eq. (11) and Lst. 3, line 13), is rather expensive since this corresponds to the "⊗" operator. We show how to reduce the number of multiplications. This optimization also holds for the simpler algorithm for $C_{\max}$ in Sec. 6.

The multiplications take place between *ranked* zeta transforms. Thus, we have $(\zeta f)(S, r) = 0$ for $|S| < r$, for any rank r. This is because when we apply the zeta transform, we first fill the rth layer of the subset lattice with the values of $f(S)$ with $|S| = r$, and then, by construction, only *supersets* will be iterated. Hence, for a rank r, sets S with $|S| < r$ will never be touched in the computation of $(\zeta f)(:, r)$.

With this observation, we can prune the range of sets S that we need to consider in line 13 (Lst. 3) even further. In the following, let $f = g$, as in the context of `DPconv`. We have:

$$\forall S.\ |S| < d \Rightarrow (\zeta f)(S, d) = 0,$$
$$\forall S.\ |S| < r - d \Rightarrow (\zeta f)(S, r - d) = 0.$$

Consequently, we can simply skip those sets S with $|S| < \max(d, r-d)$. Another further optimization is to restrict ourselves to sets S with $|S| \leq k$. This is because since we only require the kth layer, the Möbius transform only needs to consider sets of maximum cardinality k.

6 A SIMPLE ALGORITHM FOR $C_{\max}$

While our framework can support $C_{\max}$ (see Sec. 3.4), we found another a much simpler algorithm that does not require the (rather intricate) implementation of the $(\min, \max)$ semi-ring.

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 [22], 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 $(+, \cdot)$ 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 Sec. 5.

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

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:     $\gamma \leftarrow cs[p + step]$
7:     DP $\leftarrow$ LayeredDP($[c \leq \gamma]$) (Sec. 5)
8:     **if** DP($V$) $> 0$ **then**
9:         $p \leftarrow p + step$
10:    **end if**
11:    $step \leftarrow step \ / \ 2$
12: **end while**
13: **return** $cs[p]$
```

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 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). We visualize Alg. 3 in Fig. 4.

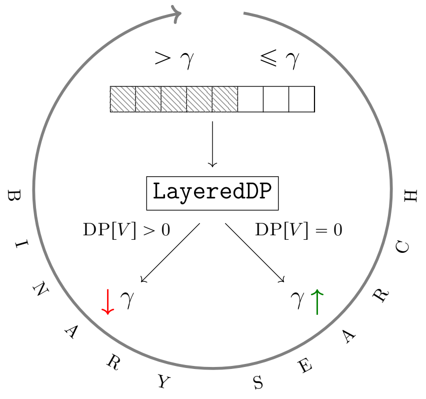

Fig. 4. Visualizing Alg. 3.

Running Time. Given our improved implementation of layered dynamic programming (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.

Constant-Factor Optimizations. While this is sufficient to outperform the standard exact algorithm, there is still an optimization that can be done that only reduces the constant factor hidden in the running time. Namely, for the first layers of the DP-table, we directly hardcode the dynamic programming solution for subsets of cardinality l.e.q. 6. This removes the overhead of subset convolution for these small layers. Note that we still need to compute the zeta transforms of these layers, since they will be used in later layers (as in Lst. 3, line 13).

7 APPROXIMATION ALGORITHM

Motivation. A prominent result on approximation algorithms for the join ordering problem is due to Chatterji et al. [5], who show that in the case of linear join trees, the problem of approximating the optimal cost K within a factor of $2^{\Theta(\log^{1-\delta} K)}$ is NP-hard, for any $\delta > 0$. We approach the problem from the other end:

How fast can we approximate the optimal C_{out} value within a factor of $(1+\varepsilon)$?

Indeed, a fast approximation algorithm can enable a faster evaluation of the optimal plan, while incurring a small overhead, specified by the precision parameter ε.

Our Approximation Algorithm. To show the benefit of reducing the problem of join ordering to subset convolution, we now show how to obtain an $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time approximation algorithm that optimizes C_{out} within a multiplicative factor of $(1+\varepsilon)$. In particular, note that our algorithm is still *exponential.* However, it shows that the $O(3^n)$-time barrier can be overcome when asking about $(1+\varepsilon)$-approximation algorithms. This is the first result of this kind, which we state in Thm. 7.2.

7.1 Approximate Min-Sum Subset Convolution

Following a recent result by Bringmann et al. [4] and the so far unexplored connection between min-plus sequence convolution and min-sum subset convolution, where the latter is the one we reduced join ordering to, Stoian [46] has shown that min-sum subset convolution can be $(1+\varepsilon)$-approximated in $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time; here, $\widetilde{O}$ hides poly-logarithmic factors in the input size and ε (note that the input size also consists of the join cardinality function, which is represented as a vector of size 2^n). For completeness, this is their main theorem:

Theorem 7.1 ([46, Thm. 3]). *$(1+\varepsilon)$-Approximate min-sum subset convolution can be solved in $\widetilde{O}(2^{\frac{3n}{2}}/\sqrt{\varepsilon})$-time.*

This result implied approximation algorithms for several problems that reduce to subset convolution, e.g., the prize-collecting Steiner tree problem [39]. Thus, by our reduction of the join ordering problem to min-sum subset convolution in Sec. 3, we can obtain an $(1+\varepsilon)$-approximation algorithm for the join ordering problem as well.

7.2 Approximate Join Ordering

The approximation algorithm follows their simple scheme:

Theorem 7.2. *If $(1+\varepsilon)$-approximate min-sum subset convolution runs in $T(n,\varepsilon)$-time, then $(1+\varepsilon)$-approximate join ordering can be solved in $O(T(n, \frac{\varepsilon}{n-1}))$-time.*

Proof. Consider the evaluation of the min-sum subset convolution between the DP-table and itself at each of the $n-1$ optimization layers; see Alg. 1, line 6. Fixing $\varepsilon' > 0$ for each convolution call, we obtain a cumulative relative error bounded by $(1+\varepsilon')^{n-1}$. By setting $\varepsilon' = \Theta(\frac{\varepsilon}{n-1})$, we obtain a relative error of at most ε. □

Corollary 7.3. *$(1+\varepsilon)$-Approximate join ordering can be solved in $\tilde{O}(2^{\frac{3n}{2}}/\sqrt{\varepsilon})$-time.*

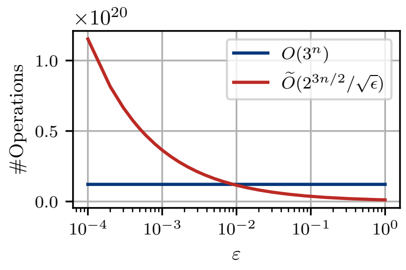

Fig. 5. Theoretical number of operations of the exact $O(3^n)$-time algorithm and the $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time $(1+\varepsilon)$-approximation algorithm for $n = 40$ and varying ε's.

In particular, we aim to optimize C_{out} (since our C_{max} algorithm in Sec. 6 already achieves a better running time). This is particularly interesting since the running time of the approximation algorithm does not depend on W, the largest join cardinality. To get an intuition for the running time of the approximation algorithm, we plot in Fig. 5 the theoretical number of operations of the exact $O(3^n)$-time algorithm and the $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time $(1+\varepsilon)$-approximation algorithm for $n = 40$ and varying ε's. For $\varepsilon = 10^{-2}$, i.e., the optimal value is approximated by a multiplicative factor of $(1 + 10^{-2})$, the runtime of the approximation algorithm outperforms that of the exact algorithm.

Note that the intricate details in the approximation framework by Bringmann et al. [4] make it hard to have an immediate practical algorithm out of the above theoretical result. We discuss this in Sec. 11.

8 FUSING C_{out} AND C_{max}

We can indeed regard the faster optimization of C_{max} from another perspective: What if we could optimize the optimal C_{out}-value under the constraint that the intermediate size is not too large? To show the motivation behind this problem, consider the optimization of C_{out} in the case of Q19d in JOB [25]. When using the true cardinalities, the max. intermediate join size of the *optimal* plan w.r.t. C_{out} is 3,036,719 tuples. In contrast, directly optimizing the largest join size via C_{max} only results in an intermediate size of 1,760,645 tuples; this reduces the largest intermediate size by 1.72x. The same can be observed in the recently introduced CEB benchmark: There is a query,[6] whose optimal C_{out} plan has the same behavior. Namely, the largest intermediate join is consists of 11,637,593 tuples, yet if we directly optimized under C_{max}, we obtain a largest intermediate join of 9,805,312 tuples.

8.1 Capping C_{out}

Having optimized for C_{max} does not represent any impediment in further refining the plan w.r.t. C_{out}. Indeed, we propose a novel cost function to be optimized for, namely the C_{cap}, motivated by the previous findings. Namely, we propose to *jointly* optimize C_{out} and C_{max}, i.e., minimize the sum of the intermediate join sizes while enforcing that the largest one is equal to the optimal C_{max} value. This ensures that we both have a bounded intermediate size (space-optimality) *and* the best time-optimal plan under this constraint.

The drawback is naturally that this joint optimization now needs two optimizer passes: (i) Find the optimal C_{max} value, and (ii) optimize C_{out} so that all intermediate join sizes are bounded above by that value. To reduce the optimization time of the second pass, we can reduce the search space

[6]Specifically, `11a/5ec72a84a33f3b3b1f4e53b734731ab0bbecebba.sql`

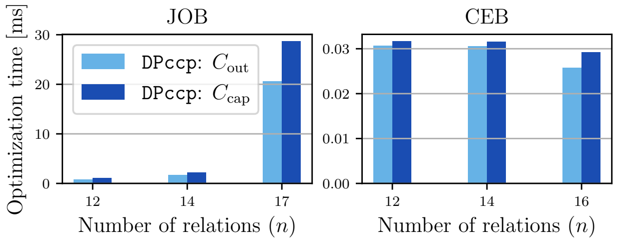

Fig. 6. Overhead in optimization time for C_{cap} on JOB [25] and CEB [32], i.e., optimizing C_{out} under the constraint that the largest intermediate size is the same as when optimizing with $C_{\max}$ (two optimization phases).

of the optimization problem, by observing that in DPccp (and DPsub) we can directly prune the intermediate solutions the size of which exceed the optimal $C_{\max}$ value.

We visualize this preliminary overhead in Fig. 6. Note that reducing this overhead is the motivation behind our novel framework, which achieves strongly-polynomial speed-up over standard join ordering algorithm, DPccp [30]. In Fig. 6, we show the price we have to pay for this joint optimization. We optimize the queries of the JOB [25] and CEB [32] benchmarks, respectively, via DPccp as follows: For C_{out}, this is the classic scenario. For C_{cap}, we first optimize $C_{\max}$ via DPccp and then run DPccp again, optimizing C_{out} under the constraint that any intermediate join size is l.e.q. the previously computed $C_{\max}$ value. In the case of JOB, for the largest join queries of 17 relations, the overhead is of 10ms. This is still negligible, but as we will show in Sec. 9.2, for larger join clique queries the overhead tends to be over 22%.

8.2 Reducing Optimization Time

The optimization of C_{cap} has in itself, first, the optimization of $C_{\max}$, and then a *pruned* C_{out} optimization. If using the standard exact join ordering algorithm, DPccp, the running time of the $C_{\max}$ optimization is still $O(3^n)$. As discussed in Sec. 6, we can reduce this running time to $O(2^n n^3)$. Therefore, we can simply use DPconv[max] and reduce the optimization time. The second pass, that of optimizing C_{out} under the constraint that the largest intermediate size does not exceed this value, remains as before. The advantage is that, since both the first pass is sped up via DPconv[max] and the second pass has a pruned search space, we show that we are even faster than a "vanilla" C_{out} optimization. We show the corresponding experiments in Sec. 9.2.

9 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 c5.xlarge EC2 instance 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. We use the setup from the CEB benchmark [32], which already provides the true cardinalities for IMDb for their 13,644 queries and the 113 queries of JOB [25] (note that this setup has already been used for Fig. 6). For clique queries, we generate random join cardinalities ≤ 100M, with the constraint that $c(S) \leq c(S_1)c(S_2), \forall S_1, S_2, \subsetneq S, S_1 \cap S_2 = \emptyset, S_1 \cup S_2 = S$, 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

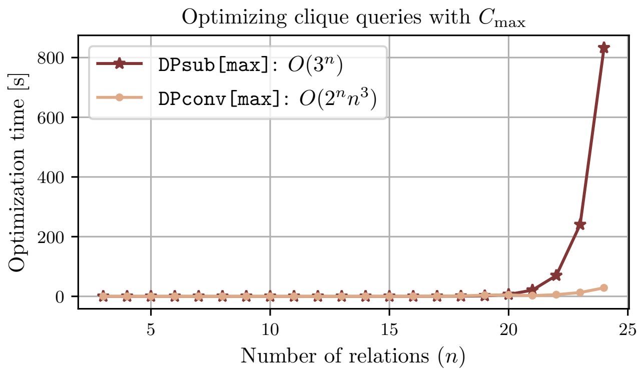

Fig. 7. Clique queries optimization: Both DPsub[max] and DPconv[max] optimize for $C_{\max}$.

optimizing for cross-products, as discussed in Sec. 3.1. Moreover, since subset convolution does not (yet) exploit sparse set functions—in our case, corresponding to unconnected query subgraphs—the running time is thus independent of the cyclicity of the query graph; we provide a discussion of this in Sec. 11.

Whenever we compare to the A*-based algorithm by Haffner and Dittrich [18], we use their benchmark set. We use the same evaluation scripts,[7] i.e., we use their generated cliques and cardinalities. We show the optimization times for cliques of up to 18 relations in Fig. 8.[8]

Competitors. The standard exact algorithms, DPccp and DPsub, follow the implementation in the reproducibility experiment of Neumann and Radke [34].[9] We also implement the bitsets as 64-bit integers, which we wrap with helper functions to provide iterators of subsets. DPconv uses all the optimizations described in Sec. 5 for layered dynamic programs. The optimization time includes the time for extracting the join tree from the layered dynamic programming.

9.1 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 $O(2^n n^2 Wn \log Wn)$-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 Sec. 6 a simple and practical algorithm for $C_{\max}$.

DPconv vs. DPsub. We benchmark on clique queries, as these are the hardest queries to optimize for [33]. 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 [30]). We show the optimization times for cliques of up to 24 relations in Fig. 7. The optimization time is averaged for each $n \in \{3, \dots, 24\}$ 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, as already shown in Fig. 6.

[7]Their reproducibility experiment is available at: https://gitlab.cs.uni-saarland.de/bigdata/mutable/evaluation
[8]In the current version, the evaluation script starts to timeout after 19 relations; we increased the default timeout to 800s, yet this did not solve the issue.
[9]https://db.in.tum.de/~radke/papers/hugejoins-reproducibility.pdf

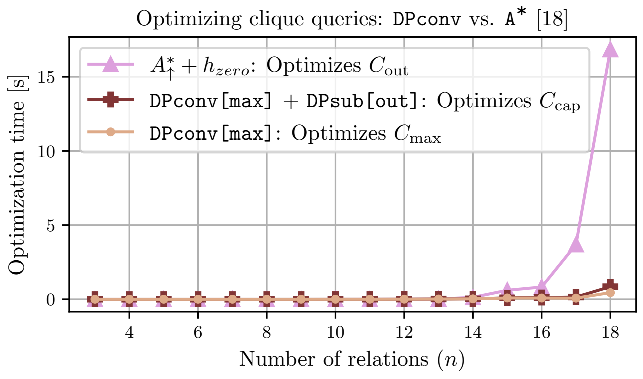

Fig. 8. Clique queries optimization (setup as in Ref. [18]): The A*-based optimizer by Haffner and Dittrich [18] optimizes for C_{out}, while `DPconv[max]` optimizes $C_{\max}$ in $O(2^n n^3)$-time, and the joint combination between `DPconv[max]` and the pruned `DPsub[out]` optimizes for C_{cap} (Sec. 8).

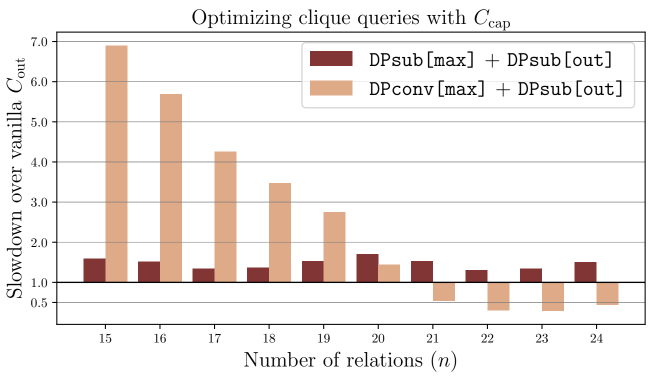

Fig. 9. The slowdown of optimizing C_{cap} for large clique queries over a "vanilla" C_{out} optimization. The baseline is `DPsub[out]`. While a naive optimization is (naturally) slower, using `DPconv[max]`, the instantiation of our novel framework for $C_{\max}$, in the first optimization pass, and followed by the pruned C_{out} optimization, we obtain an optimization time even faster than that of a "vanilla" C_{out}.

DPconv vs. A*. Haffner and Dittrich [18], as part of their mu*t*able database system, have recently shown that, indeed, using an A^*-based optimizer, one can reduce the number of ccp to be explored. Note that this indeed leads to an optimal solution. In particular, unlike `DPconv[max]`, the practical instantiation of `DPconv` for $C_{\max}$, their algorithm optimizes for C_{out}. Thus, the following experiment solely serves to compare the running times, as the cost functions to be optimized are different.

9.2 Optimizing C_{cap}

We show that the optimization time of C_{cap} can be made practical using our novel `DPconv` framework. Recall that optimizing C_{cap} requires two optimization passes. We will focus on the first pass, in which we optimize for $C_{\max}$. The reason is that we can use `DPconv[max]`, the instantiation of our novel framework `DPconv` for the $C_{\max}$ cost function. This reduces the running time of this pass from $O(3^n)$-time to $O(2^n n^3)$-time. This is particularly significant for large join queries.

To this end, in Fig. 9, we show the slowdown of optimizing C_{cap} compared to a "vanilla" C_{out} optimization. We benchmark on clique queries, as previously argued. To not clutter the plot, we only keep DPsub as the baseline for clique queries. Thus, the baseline is the optimization of C_{cap} via DPsub, namely we first optimize $C_{\max}$ and then run a pruned C_{out} optimization, i.e., we then skip the subsets whose intermediate size is larger than this latter value. Our proposed algorithm replaces DPsub[max] with DPconv[max] in the first pass. We first observe that, naturally, the naïve C_{cap} optimization is slower than the "vanilla" C_{out} optimization (slow-down is over 22%). In contrast, having both a super-polynomial speedup for the first optimization due to DPconv and a pruned search space for the second pass, we are even faster than the "vanilla" C_{out} optimization after 21 relations. Compared to the A*-based algorithm, the optimization of C_{cap} outperforms that of C_{out} after 14 relations as well.

Analyzing C_{cap} on CEB. Out of the 13,644 queries of the CEB [32] benchmark, there are 2,873 queries for which the largest intermediate size in the optimal C_{out} plan is 6.8% larger than the optimal $C_{\max}$ intermediate size. For these queries, $C_{\max}$ looses 22.8% in the optimal C_{out} value, while C_{cap} naturally reduces this to only 9.5%.

10 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.

10.1 Exact Algorithms

The history of the join ordering problem starts at Selinger, proposing an $O(4^n)$-time algorithm, commonly referred to as DPsize [44]. 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 [49] 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 [36] analyzed the *minimum* number of subplan pairs that have to be iterated in any dynamic program. To this end, Moerkotte and Neumann [30] 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 [18] 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 $\widetilde{O}(2^n W)$-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 [16]. Hence, Chaudhuri et al. [6] explore the possiblity of implementing join ordering as a top-down procedure, only considering linear solutions. Building on this work, DeHaan and Tompa [10] extend the top-down method to bushy join trees, disallowing cross products. Fender and Moerkotte [14, 15] improve the running time of these algorithms and get rid of the connectedness check, i.e., only outputting the ccp's.

10.2 Approximation Algorithms

Exact algorithms are rather expensive. To this end, Chatterji et al. [5] analyzed whether there are instances that can be solved by approximation algorithms in polynomial time. Unless P = 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 from Sec. 7 runs in *exponential* time.)

10.3 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 [20, 24]. Neumann and Radke [34] 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) [13] 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 [23] 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 [45], mixed-integer linear programming [47], and simulated annealing [45]. 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 [42, 51]. 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 [28] 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 [41], a further promising direction is query super-optimization [27].

11 DISCUSSION

Resource-Aware Query Optimization. The trend nowadays is to execute queries in multi-tenant cloud machines. Recently, Viswanathan et al. [50] 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 optimization of C_{out} and $C_{\max}$ can go beyond our proposed cost function C_{cap}. With C_{cap}, we first compute the optimal value of $C_{\max}$ and then do a pruned C_{out} optimization. Instead of taking the *optimal* $C_{\max}$ value, capping C_{out} at the 90th percentile of the largest intermediate size allows for more flexibility. So one can effectively trade off between query runtime and memory consumption. This is particularly interesting in cloud scenarios.

The cloud data warehouse Amazon Redshift uses predicted query memory to make scheduling decisions [40]. 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 want to 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 want to produce a high memory but fast-running query.

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 (Sec. 3.3) and the approximation algorithm (Sec. 7). In particular, the details of the framework by Bringmann et al. [4], upon which the approximate min-sum subset convolution algorithm is based on, span several pages.

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 [21] for the latest results on sparse sequence convolution). This would be particularly useful for sparse query graphs of the JOB [25] and CEB [32] benchmarks. These queries do not benefit from the speedup obtained by DPconv due to the fact they only touch at most 17 relations. An algorithmic advance in subset convolution for the sparse setting can be directly transferred to the join ordering problem.

12 CONCLUSION

Join ordering, or finding the optimal order of the joins in a query, is an indispensable task in a database management system. The problem has its roots in the seminal work of Selinger [44], culminating with the graph-theoretic exact algorithm by Moerkotte and Neumann [30]. Despite recent research [18], the worst-case running time still remains $O(3^n)$.

In this work, we provided the first super-polynomial speedup over the standard dynamic programming solution. Our framework optimizes (i) C_{out} in $\widetilde{O}(2^n)$-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 [9], and uses the fact that join ordering is implicitly a dynamic programming recursion using subset convolution similar to other classic problems in the literature (see Björklund et al. [2]). The reduction to subset convolution also implies an $(1+\varepsilon)$-approximation algorithm for optimizing C_{out} in $\widetilde{O}(2^{3n/2}/\sqrt{\varepsilon})$-time.

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. In addition, we showed that joint optimization of C_{out} and $C_{\max}$ results in faster optimization times than a "vanilla" C_{out} after 21 relations, while only increasing C_{out} by 9.5%.

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

REFERENCES

[1] R. Bellman, R.E. Bellman, and Rand Corporation. 1957. *Dynamic Programming*. Princeton University Press. https://books.google.ro/books?id=rZW4ugAACAAJ

[2] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. 2007. Fourier meets möbius: fast subset convolution. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, David S. Johnson and Uriel Feige (Eds.). ACM, 67–74. https://doi.org/10.1145/1250790.1250801

[3] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. 2009. Set Partitioning via Inclusion-Exclusion. *SIAM J. Comput.* 39, 2 (2009), 546–563. https://doi.org/10.1137/070683933

[4] Karl Bringmann, Marvin Künnemann, and Karol Wegrzycki. 2019. Approximating APSP without scaling: equivalence of approximate min-plus and exact min-max. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, Moses Charikar and Edith Cohen (Eds.). ACM, 943–954. https://doi.org/10.1145/3313276.3316373

[5] Sourav Chatterji, Sai Surya Kiran Evani, Sumit Ganguly, and Mahesh Datt Yemmanuru. 2002. On the Complexity of Approximate Query Optimization. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis (Eds.). ACM, 282–292. https://doi.org/10.1145/543613.543650

[6] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, Philip S. Yu and Arbee L. P. Chen (Eds.). IEEE Computer Society, 190–200. https://doi.org/10.1109/ICDE.1995.380392

[7] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[8] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[9] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. 2015. *Algebraic techniques: sieves, convolutions, and polynomials.* Springer International Publishing, Cham, 321–355. https://doi.org/10.1007/978-3-319-21275-3_10

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

[11] Stuart E. Dreyfus and Robert A. Wagner. 1971. The steiner problem in graphs. *Networks* 1, 3 (1971), 195–207. https://doi.org/10.1002/NET.3230010302

[12] Marius Eich, Pit Fender, and Guido Moerkotte. 2018. Efficient generation of query plans containing group-by, join, and groupjoin. *VLDB J.* 27, 5 (2018), 617–641. https://doi.org/10.1007/S00778-017-0476-3

[13] Leonidas Fegaras. 1998. A New Heuristic for Optimizing Large Queries. In *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1460)*, Gerald Quirchmayr, Erich Schweighofer, and Trevor J. M. Bench-Capon (Eds.). Springer, 726–735. https://doi.org/10.1007/BFB0054528

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

[15] Pit Fender and Guido Moerkotte. 2012. Reassessing Top-Down Join Enumeration. *IEEE Trans. Knowl. Data Eng.* 24, 10 (2012), 1803–1818. https://doi.org/10.1109/TKDE.2011.235

[16] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. 2012. Effective and Robust Pruning for Top-Down Join Enumeration Algorithms. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 414–425. https://doi.org/10.1109/ICDE.2012.27

[17] Fedor V. Fomin and Dieter Kratsch. 2010. *Exact Exponential Algorithms* (1st ed.). Springer-Verlag, Berlin, Heidelberg.

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

[19] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4, 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[20] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Trans. Database Syst.* 9, 3 (1984), 482–502. https://doi.org/10.1145/1270.1498

[21] Ce Jin and Yinzhan Xu. 2024. Shaving Logs via Large Sieve Inequality: Faster Algorithms for Sparse Convolution and More. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada,*

June 24-28, 2024, Bojan Mohar, Igor Shinkar, and Ryan O'Donnell (Eds.). ACM, 1573–1584. https://doi.org/10.1145/3618260.3649605

[22] S.R. Kosaraju. 1989. Efficient tree pattern matching. In *30th Annual Symposium on Foundations of Computer Science*. 178–183. https://doi.org/10.1109/SFCS.1989.63475

[23] Donald Kossmann and Konrad Stocker. 2000. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.* 25, 1 (2000), 43–82. https://doi.org/10.1145/352958.352982

[24] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. 1986. Optimization of Nonrecursive Queries. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 128–137. http://www.vldb.org/conf/1986/P128.PDF

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

[26] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, Haran Boral and Per-Åke Larson (Eds.). ACM Press, 18–27. https://doi.org/10.1145/50202.50204

[27] Ryan Marcus. 2023. Learned Query Superoptimization. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023 (CEUR Workshop Proceedings, Vol. 3462)*, Rajesh Bordawekar, Cinzia Cappiello, Vasilis Efthymiou, Lisa Ehrlinger, Vijay Gadepally, Sainyam Galhotra, Sandra Geisler, Sven Groppe, Le Gruenwald, Alon Y. Halevy, Hazar Harmouch, Oktie Hassanzadeh, Ihab F. Ilyas, Ernesto Jiménez-Ruiz, Sanjay Krishnan, Tirthankar Lahiri, Guoliang Li, Jiaheng Lu, Wolfgang Mauerer, Umar Farooq Minhas, Felix Naumann, M. Tamer Özsu, El Kindi Rezig, Kavitha Srinivas, Michael Stonebraker, Satyanarayana R. Valluri, Maria-Esther Vidal, Haixun Wang, Jiannan Wang, Yingjun Wu, Xun Xue, Mohamed Zaït, and Kai Zeng (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-3462/AIDB5.pdf

[28] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, Rajesh Bordawekar and Oded Shmueli (Eds.). ACM, 3:1–3:4. https://doi.org/10.1145/3211954.3211957

[29] Guido Moerkotte. 2023. *Building Query Compilers (Draft / Under Construction).* https://pi3.informatik.uni-mannheim.de/%7Emoer/querycompiler.pdf

[30] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 930–941. http://dl.acm.org/citation.cfm?id=1164207

[31] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 539–552. https://doi.org/10.1145/1376616.1376672

[32] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[33] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 403–414. https://doi.org/10.1145/1559845.1559889

[34] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 677–692. https://doi.org/10.1145/3183713.3183733

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

[36] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek (Eds.). Morgan Kaufmann, 314–325. http://www.vldb.org/conf/1990/P314.PDF

[37] Oriana Ponta, Falk Hüffner, and Rolf Niedermeier. 2008. Speeding up Dynamic Programming for Some NP-Hard Graph Recoloring Problems. In *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4978)*, Manindra Agrawal, Ding-Zhu Du, Zhenhua Duan, and Angsheng Li (Eds.). Springer, 490–501. https://doi.org/10.1007/978-3-540-79228-4_43

[38] Bernhard Radke and Thomas Neumann. 2019. LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings (LNI, Vol. P-289)*, Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer (Eds.). Gesellschaft für Informatik, Bonn, 57–76. https://doi.org/10.18420/BTW2019-05

[39] Daniel Rehfeldt and Thorsten Koch. 2022. On the Exact Solution of Prize-Collecting Steiner Tree Problems. *INFORMS J. Comput.* 34, 2 (2022), 872–889. https://doi.org/10.1287/IJOC.2021.1087

[40] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 225–237. https://doi.org/10.1145/3555041.3589677

[41] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 347–359. https://doi.org/10.1145/3626246.3653395

[42] Manuel Schönberger, Stefanie Scherzinger, and Wolfgang Mauerer. 2023. Ready to Leap (by Co-Design)? Join Order Optimisation on Quantum Hardware. *Proc. ACM Manag. Data* 1, 1 (2023), 92:1–92:27. https://doi.org/10.1145/3588946

[43] Jacob Scott, Trey Ideker, Richard M. Karp, and Roded Sharan. 2005. Efficient Algorithms for Detecting Signaling Pathways in Protein Interaction Networks. In *Research in Computational Molecular Biology, 9th Annual International Conference, RECOMB 2005, Cambridge, MA, USA, May 14-18, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3500)*, Satoru Miyano, Jill P. Mesirov, Simon Kasif, Sorin Istrail, Pavel A. Pevzner, and Michael S. Waterman (Eds.). Springer, 1–13. https://doi.org/10.1007/11415770_1

[44] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[45] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. 1997. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB J.* 6, 3 (1997), 191–208. https://doi.org/10.1007/S007780050040

[46] Mihail Stoian. 2024. Sinking an Algorithmic Isthmus: (1 + epsilon)-Approximate Min-Sum Subset Convolution. arXiv:2404.11364 [cs.DS]

[47] Immanuel Trummer and Christoph Koch. 2017. Solving the Join Ordering Problem via Mixed Integer Linear Programming. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1025–1040. https://doi.org/10.1145/3035918.3064039

[48] Bennet Vance. 1998. *Join-order Optimization with Cartesian Products.* Ph. D. Dissertation. Oregon Graduate Institute of Science and Technology.

[49] Bennet Vance and David Maier. 1996. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 35–46. https://doi.org/10.1145/233269.233317

[50] Lalitha Viswanathan, Alekh Jindal, and Konstantinos Karanasos. 2018. Query and Resource Optimization: Bridging the Gap. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1384–1387. https://doi.org/10.1109/ICDE.2018.00156

[51] Tobias Winker, Umut Çalikyilmaz, Le Gruenwald, and Sven Groppe. 2023. Quantum Machine Learning for Join Order Optimization using Variational Quantum Circuits. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE 2023, Seattle, WA, USA, 18 June 2023*, Sven Groppe, Le Gruenwald, and Ching-Hsien Hsu (Eds.). ACM, 5:1–5:7. https://doi.org/10.1145/3579142.3594299

[52] Frank Yates. 1937. The Design and Analysis of Factorial Experiments. *Imperial Bureau of Soil Science* (1937).

Received April 2024; revised July 2024; accepted August 2024