LinDP++: Generalizing Linearized DP to Crossproducts and Non-Inner Joins

Bernhard Radke[1] Thomas Neumann[2]

Abstract: Choosing the best join order is one of the main tasks of query optimization, as join ordering can easily affect query execution times by large factors. Finding the optimal join order is NP-hard in general, which means that the best known algorithms have exponential worst case complexity. As a consequence only relatively modest problems can be solved exactly, which is a problem for today's large, machine generated queries. Two developments have improved the situation: If we disallow crossproducts, graph-based DP algorithms have pushed the boundary of solvable problems to a few dozen relations. Beyond that, the linearized DP strategy, where an optimal left-deep plan is used to linearize the search space of a subsequent DP, has proven to work very well up to a hundred relations or more.

However, these strategies have limitations: Graph-based DP intentionally does not consider implicit crossproducts, which is almost always ok but sometimes undesirable, as in some cases such crossproducts are beneficial. Even more severe, linearized DP can handle neither crossproducts nor non-inner joins, which is a serious limitation. Large queries with, e. g., outer joins are quite common and having to fall back on simple greedy heuristics in this case is highly undesirable.

In this work we remove both limitations: First, we generalize the underlying linearization strategy to handle non-inner joins, which allows us to linearize the search space of arbitrary queries. And second, we explicitly recognize potential crossproduct opportunities, and expose them to the join ordering strategies by augmenting the query graph. This results in a very generic join ordering framework that can handle arbitrary queries and produces excellent results over the whole range of query sizes.

1 Introduction

One of the most important tasks of query optimization is join ordering. Due to the multiplicative nature of joins, changes in join order can easily affect query execution times by large integer factors [Le18]. Unfortunately, finding the optimal join order is NP-hard in general [IK84] and no exact algorithms with better than exponential worst case optimization time are known for the general case. This is problematic because queries tend to get larger, at least in the long tail. Today, most queries are not written by humans but by machines, and queries that join a hundred relations or more are not that uncommon [Vo18]. To put that into perspective, PostgreSQL for example switches from dynamic programming (DP) to a

[1] Technische Universität München, radke@in.tum.de
[2] Technische Universität München, neumann@in.tum.de

cc by nc

heuristic if the query contains 12 relations or more, which means that none of the larger queries will be optimized exactly.

One reason for that is their somewhat simplistic DP strategy. DP strategies that exploit the structure of the query graph for example can handle larger queries [MN08], but even there the exponential nature of the problem limits query sizes to about 30 relations, depending upon on the structure of the query graph. For even larger queries most approaches fall back to simple heuristics. An alternative to that is the relatively recent *linearized DP* strategy [NR18]. The idea is to linearize the search space by first picking a good (ideally optimal) relative order of relations, and then use a polynomial time DP step to construct the optimal bushy tree for that relative order. Of course we do not know the optimal order for the general solution, but we can use the IK/KBZ algorithm [IK84, KBZ86] to construct the optimal left-deep order in polynomial time. In practice this leads to excellent results, producing optimal or near-optimal solutions even for large queries with very low optimization time.

However, the IK/KBZ algorithm supports only inner joins, which is a problem for practical usage. Outer, semi, and anti joins are quite common: In the real-world workload presented by Vogelsgesang et al. [Vo18], about 20% of the join queries do contain at least one outer join with a maximum of 247 outer joins in a single query. Having to fall back to simple heuristics just because the query contains a single outer join is not very satisfying. Similar problems occur with complex predicates, for example predicates of the form $R_1.A + R_2.B = R_3.C + R_4.D$. These complex predicates are rare, but they can be formulated in SQL. In the query graph they form a hyperedge, connecting sets of relations with sets of relations, which is also not supported by IK/KBZ. Note that non-inner joins can be expressed by using hyperedges, too [MFE13], thus both problems are closely related from an optimizer perspective. These restrictions are very unfortunate, as now queries with simple inner joins can be optimized very efficiently, but adding just one non-inner join or one complex join predicate forces the system to switch to simple heuristics, resulting in clearly inferior plans.

Furthermore, graph-based DP as well as linearized DP ignore crossproducts. Usually this is a good idea. The search space without crossproducts is much smaller, and in most cases crossproducts are a bad idea. However, sometimes they can indeed be helpful if some input relations or intermediate results are known to be very small. Even then, crossproducts should be used prudently as mis-estimations about input cardinalities can lead to terrible execution times due to the $O(n^2)$ nature of a crossproduct. And considering crossproducts in the presence of non-inner joins is dangerous as that can lead to wrong results. Consider e. g. the query $(A \bowtie B) ⟕_{A.x=C.y} C$ and assume $B = \emptyset$. Performing a crossproduct between B and C before evaluating the outer join would cause an empty result, whereas the original query yields the complete relation C. Nevertheless, if we make sure that a crossproduct does not bypass non-inner joins and we are certain about the input cardinalities (e. g. when a primary key is bound), crossproducts can sometimes significantly improve query performance [OL90]. Having support for crossproducts in “safe” cases is thus highly desirable.

In this paper we generalize the recently published linearized DP [NR18] by removing both limitations. Our generalized LinDP++ strategy is capable of ordering non-inner joins, which allows it to handle all kinds of join queries. We achieve this using a recursive precedence-graph decomposition at hyperedges, which allows IK/KBZ to handle hypergraphs. In addition we present a fast heuristic that explicitly enriches the search space to also consider safe crossproducts without causing the search space to grow exponentially. The combination of these two components results in a fast polynomial time heuristic for join ordering that finds very good plans, explicitly investigates relevant crossproducts, and handles non-inner joins correctly. Experimental comparisons with slow exact DP strategies show that the resulting plans are close to optimal. And the algorithm can scale to queries with a hundred relations or more, which is far beyond what normal DP algorithms can do. For practical usage this is a great improvement, as we no longer have to fall back to weaker approaches for certain classes of queries.

The rest of this paper is structured as follows: First we summarize prior work in Section 2. The extension of linearized DP to non-inner joins is described in Section 3. In Section 4 we investigate how join ordering can be extended to take beneficial crossproducts into consideration. We evaluate runtime characteristics and result quality of LinDP++ in Section 5 before we draw a conclusion and point out directions for future research in Section 6.

2 Related Work

Join ordering has first been tackled by Selinger et al. in [Se79]. They proposed a dynamic programming (DP) strategy that generates an optimal linear join tree. Optimal solutions for subproblems of increasing size are built bottom up by combining optimal solutions for smaller subproblems. Since then there has been lots of follow-up work of which we discuss the most relevant techniques in the following.

An obvious improvement over the initial DP is to consider bushy trees as well. Furthermore, for a DP algorithm to work it is not necessary to enumerate alternatives increasing in size. Other enumeration schemes work as well (e. g. integer order enumeration [VM96]), as long as optimal solutions for subproblems are generated prior to their usage in larger problems. All of these dynamic programming variants can be implemented to consider crossproducts. In this case, however, *all* possible crossproducts would implicitly be enumerated. This results in exponential complexity of the algorithms and disregards the actual structure of the query. In addition to this increase in complexity, crossproducts between arbitrary relations can produce incorrect query results in the presence of outer joins.

The most efficient DP algorithms take the query graph into account. By design, such algorithms generate only execution plans without crossproducts. With the reordering constraints induced by outer joins encoded into the query graph, they can also validly reorder across outer joins [MN08]. Besides bottom-up enumeration there also exist variants that perform the enumeration top-down, thereby enabling more aggressive pruning [FM13]. As

these algorithms strictly follow the structure of the query graph, they per se do not generate crossproducts. crossproducts can, however, explicitly be taken into account by adding edges to the query graph and updating the reordering constraints of outer joins.

Another approach to incorporate the reordering constraints imposed by non-inner joins is to add compensation operators [WC18]. These operators correct errors introduced by crossproducts across outer joins by removing or modifying spurious tuples. Materializing these spurious tuples and manipulating them, however, creates overhead at query runtime.

Hardware trends have motivated research on parallelizing dynamic programming [Ha08]. However, linearly increasing the compute power cannot compensate for the exponential growth of the search space. Doubling e. g. the compute power only allows queries with one additional join to be optimized exactly in a similar, reasonable amount of time.

Lately, the use of linear programming for join ordering has been proposed [TK17]. The mixed integer linear program (MILP) that they generate encodes relations, cardinalities, and costs. The solution of the MILP can then be interpreted as a linear join tree. As they do not fully constrain the MILP to the query graph, the solution may contain crossproducts. For queries with outer joins this can again lead to invalid execution plans.

Many commercial systems find a good join order by applying transformations onto the initial execution plan [Gr95]. These transformative approaches have the advantage that relational equivalences can directly be translated into transformation rules. Direct application of equivalences enables the algorithms to take care of reordering constraints as transformation rules can be disabled as required. However, these algorithms are considerably slower than DP style algorithms and avoiding to generate trees multiple times is non-trivial. These issues become even more prominent if additional rules to generate crossproducts are introduced.

Besides exact algorithms that give an optimal tree, a large number of heuristics have been proposed especially to handle large queries. One well known heuristic is the genetic algorithm [SMK97], a variant of which is used by PostgreSQL for queries containing more than 12 joins. The genetic meta-heuristic starts with a population of randomly generated execution plans. For a number of generations, crossover and mutation is applied and the best plans of the resulting population survive the generation.

Another interesting algorithm is Greedy Operator Ordering (GOO) [Fe98]. This heuristic builds a bushy join tree bottom up by picking the pair of relations whose join result is minimal. The picked pair is then merged into a single relation representing the join. Repeated application of this procedure finally results in a single relation representing a complete join tree. GOO is fast even for large queries and usually gives decent plans despite it's greediness.

Iterative Dynamic Programming (IDP) [KS00] is a combination of DP and GOO. It has proven to work well especially for really large queries. By using DP to refine expensive parts of a join tree generated by a greedy algorithm, plan costs can be significantly reduced.

In [NR18] we described linearized DP, a heuristic for join ordering on large queries. We avoid the exponential complexity of a full dynamic programming strategy by restricting the DP algorithm to a reduced, linearized search space. Utilizing this technique we bridged the gap between the small queries which can be optimized exactly and the really large queries, where only greedy heuristics have acceptable runtime. While this approach gives excellent results for regular queries, its inability to handle outer joins is a major drawback which we tackle in this paper.

3 Search Space Linearization

Dynamic Programming (DP) algorithms can be used to solve the join ordering problem exactly, but the exponential worst case complexity of all known algorithms limits their use to relatively small queries. The exact complexity depends upon the structure of the explored search space: A join query Q induces an undirected query graph $G = (V, E)$, where V is the set of relations, and E is the set of join possibilities between relations. In the general case, or when the query graph forms a clique, the best known algorithm has a time complexity in $O(3^n)$, which is infeasible for large n. But when the query graphs forms a linear chain and if crossproducts are not allowed, the optimal solution can be found in $O(n^3)$ [MN06], which is tractable even for large n.

This observation recently led to the concept of linearized DP [NR18]. The key idea is as follows: Assume that we would know the optimal join order. Then we could take the relative order of the relations in the optimal join tree and linearize the search space by restricting the DP algorithm to consider only sub-chains of that relative order. Given the optimal relative order as input the DP phase can construct the optimal bushy tree in $O(n^3)$ [NR18].

Of course we do not know the optimal join order, and thus we do not know the optimal relative order, either. But for a large class of queries we can construct the optimal left-deep tree in polynomial time using the IK/KBZ algorithm [IK84, KBZ86]. This gives us a relative order of relations, too, which we can then use for search space linearization. Using the IK/KBZ solutions as seed for search space linearization is a heuristic, as we can cut the optimal solution from the search space, but 1) the resulting plan is never worse than the optimal left-deep plan, and 2) it is usually close to the true optimal solution in practice, with much lower optimization time [NR18]. Note that while IK/KBZ requires the cost function to have Adjacent Sequence Interchange (ASI) property [MS79], the subsequent DP phase can use any cost function that adheres to the bellman principle.

The main limitation of this technique is that IK/KBZ cannot handle arbitrary queries. First, it requires an acyclic query graph. We can avoid that problem by first constructing a minimum spanning tree before executing IK/KBZ. The intuition behind this is that less selective joins are less likely to be part of the optimal join tree, thus dropping edges that correspond to such joins to break cycles is usually safe. Note that the DP phase of the algorithm can again operate on the original, complete query graph potentially including

Algorithm 1 The IKKBZ algorithm [IK84, KBZ86]

```
IKKBZ(Q = (V, E))
// construct an optimal left-deep tree for the acyclic query graph Q
  b = ∅
  for each s ∈ V
    P_v = IKKBZ-precedence(Q, s, ∅)
    while P_v is not a chain
      pick v′ in P_v whose children are chains
      IKKBZ-normalize each child chain of v′
      merge the child chains by rank
    if b = ∅ ∨ C(P_v) < C(b)
      b = P_v
  return b

IKKBZ-precedence(Q(V, E), v, X)
// build a precedence tree by directing edges away from a node v ∈ V
  P_v = v
  for each e ∈ E : (e = (v, u) ∨ e = (u, v)) ∧ u ∉ X
    add IKKBZ-precedence(Q, u, X + v) as child of P_v
  return P_v

IKKBZ-normalize(c)
//normalize a chain of relations
  while ∃i: rank(c[i])>rank(c[i + 1])
    merge c[i] and c[i + 1] into a compound relation
```

cycles. More severely, IK/KBZ cannot handle hyperedges in the query graph, which would be necessary to support non-inner joins and complex join predicates. For example the join query $(A \bowtie B) ⟗ (C \bowtie D)$ will have the regular edges (A, B), (C, D), and the hyperedge $(\{A, B\}, \{C, D\})$, which captures the reordering constraints of inner joins and outer joins. Note that this is a fundamental problem: The algorithm constructs left-deep trees, but that query graph has no valid left-deep solution, all solutions must be bushy. In order to apply the idea of search space linearization to queries with non-inner joins we must therefore extend IK/KBZ to handle hyperedges.

In the following we first briefly repeat how regular search-space linearization works, and then show an extension to handle hyperedges.

3.1 Regular Search Space Linearization

Before discussing the linearization for hypergraphs, let us briefly reiterate, how the linearization of regular graphs works. The IK/KBZ algorithm [IK84, KBZ86] constructs an optimal left-deep join tree, which is then used as relative relation order in the linearized DP.

As input the algorithm gets an acyclic query graph. For cyclic query graphs we construct a minimum spanning tree first.

The pseudo-code for the IK/KBZ algorithm is shown in Algorithm 1. It chooses each relation s as start node once, and then runs the complete construction algorithm given that start node. For each s, it first constructs the directed precedence graph P_v rooted in s by directing all join edges away from s. That precedence graph indicates which relations have to be joined first before other joins become feasible. That is, all valid join orders adhere to the partial order induced by the precedence graph. Then the algorithm tries to sort all relations by the cost/benefit ratio of performing the join with a relation. This ratio is called rank [IK84]. If we get conflicts between the rank order and the order imposed by the precedence graph, i. e., if we would like to join with R_1 before R_2, but the precedence graph requires that R_2 is joined before R_1, the `IKKBZ-normalize` function takes both relations and combines them into a compound relation, because we know that both must occur next to each other in the optimal solution [IK84]. The remaining relations are ordered by rank until we get a total order.

The traditional IK/KBZ algorithm returns the cheapest of these n total orders, which is guaranteed to be an optimal left-deep execution plan. For the linearized DP we take the total order and use it to restrict the search space considered by the DP phase [NR18]. Note that we get better results by running the DP phase not only on the order in the cheapest plan, but on all P_v orders. The reason for that is that the optimal bushy order can be different from the optimal left-deep order. The different P_v orders are the optimal left-deep orders given a certain start node; by considering all of them we give the DP algorithm a chance to recognize orders that are more expensive left-deep but cheaper in bushy form.

3.2 Precendence for Hypergraphs

The IK/KBZ algorithm is only capable of producing linearizations for regular, acyclic graphs. When generalizing it to handle hypergraphs we first have to construct a precedence graph, too, which is a bit non-intuitive for hyperedges. The hyperedges have to be directed away from the start node, but note that a hyperedge connects a set of relations with a set of relations. To express that, a directed hyperedge is defined similar to the definition used by Gallo et al. [GLP93]:

Definition 1. *In a directed hypergraph $H = (V, E)$, a directed edge e from $T \subseteq V$ to $H \subseteq V$ is an ordered pair $e = (T, H)$, where T is said to be the tail and H the head of the edge.*

We differentiate two types of edges during precedence graph construction: backward hyperedges $b = (T, H) : |T| > 1$ and forward hyperedges $f = (T, H) : |H| > 1$. Note that an edge can be both a backward and a forward edge.

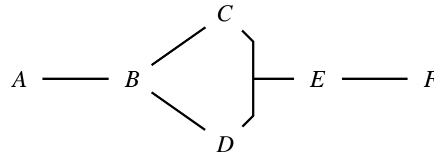


Fig. 1: A query graph with a hyperedge $(\{C, D\}, \{E\})$

For backward hyperedges all relations in the tail set T have to be available before any relation of the head H can be joined. Such edges, thus, have to be postponed until all tail relations are covered by the precedence graph. If all relations in T lie on a single path from the start relation s to the last visited relation of T we can simply append the backward edge to that relation, because we know that all other relations must have been joined before. If that is not the case, i. e., if some relations lie on different paths from the start relation we still attach the edge to the last visited relation of T, but we mark it as *partial*. We cannot statically guarantee that all relations in T will be available when considering the join and must re-check that when merging child chains.

Consider for example the query graph shown in Fig. 1. When building the precedence graph rooted in B, the backward hyperedge $(\{C, D\}, \{E\})$ has to be handled. If w.l.o.g D is visited after C during construction, then E is added as a child of D. Note that E is partial here, as it additionally requires C, which is not part of the path from B to D. All other edges are regular and handled as in IK/KBZ which results in the precedence graph given in Fig. 2.

For forward hyperedges, all relations in the head set H must be available on the right-hand side of the join. In particular, there exists no left-deep solution, the final solution must be bushy and contain a join with a super-set of H on the right hand side. The key insight here is that the query graph has to be acyclic anyways to apply IK/KBZ. Thus, if we cut the graph at the forward hyperedge we get exactly two disconnected sub-graphs, which can be optimized independently. We call the head of such a forward edge a *group* and optimize it recursively when encountering a forward edge during precedence graph construction. Note that the solution of a group is independent of the currently investigated start relation and can therefore be reused across start relations. When integrating the recursive solution into the precedence graph we could add all relations in the sub-graph as one compound relation, but that would be overly restrictive. Instead, only the minimal subchain that covers the groups relations is added as compound relation and the rest is kept as individual relations. Due to the recursive nature of this scheme, individual relations here can be compound relations from other hyperedges, of course.

An example of a precedence graph dealing with a forward hyperedge is shown in Fig. 3. This is again a precedence graph for the query in Fig. 1, this time rooted in E. When the forward hyperedge $(\{E\}, \{C, D\})$ is encountered, the group $\{C, D\}$ has to be solved. The solution of the group, which covers at least the relations B, C and D, forms a compound

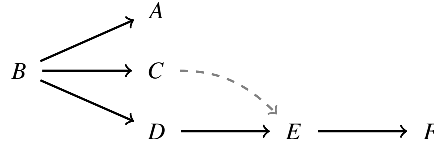


Fig. 2: Generalized precedence hypergraph of the graph shown in Fig. 1 rooted in B. E is partial and additionally requires C.

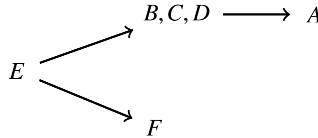


Fig. 3: Generalized precedence hypergraph of the graph shown in Fig. 1 rooted in E. $\{B, C, D\}$, the compound solution to the group of the forward hyperedge $(\{E\}, \{C, D\})$ is appended to E

node and is appended as child of E. Note that the solution to $\{C, D\}$ would additionally include A if $\text{rank}(A) < max(\text{rank}(B), \text{rank}(C), \text{rank}(D))$.

If an edge is both, a backward and a forward hyperedge, both strategies are combined: Application of the edge has to be postponed until the complete tail is available and the solution for the head group must be inserted when the tail is completely included.

3.3 Linearization using Precedence Hypergraphs

Using the generalized precedence graph we can now run a modified IK/KBZ algorithm to find a linearization. Similar to the original IK/KBZ algorithm, this is achieved by merging the nodes in subchains ascending in their rank. One difference is that nodes might already be compound relations (if forward-edges are encountered), but that does not require code changes. Backward edges are more difficult, as they have to be recognized during normalization: A sequence AB must not be normalized if B is partial, as this would prevent interleaving other nodes that are required by B between A and B. Instead, the nodes are kept separate in the precedence graph. The rank in the subchain of B is no longer monotonic here, which requires some care during implementation, but in practice B is merged as soon as possible after A.

The modifications to IK/KBZ making it hypergraph aware are given in Algorithm 2, where IKKBZ-precedence is called as IKKBZ-precedence$(Q, \emptyset, \{s\}, \emptyset)$ for each start node s.

As an example, let us now linearize the search space of the query depicted in Fig. 1 for start relation E (cardinalities and selectivities are given in Tab. 1). The algorithm starts

Algorithm 2 IKKBZ procedures generalized to hypergraphs

IKKBZ-solve-group($Q(V,E), I, G, X$)
// solve a group G
  if $|G| = 1$ **return** G
  if memoized(G) **return** memoized(G)
  $b = \emptyset$
  for each $s \in G$
    P_v = IKKBZ-precedence($Q, \emptyset, s, X \cup I + s$)
    while P_v is not a chain
      pick v' in P_v whose children are chains
      IKKBZ-normalize each child chain of v'
      merge the child chains by rank
    if $b = \emptyset \vee C(P_v) < C(b)$
      $b = P_v$
  r = smallest subsequence of b that covers all $g \in G$
  memoize r as solution for G
  return r

IKKBZ-precedence($Q(V,E), I, G, X$)
// build a precedence tree by directing edges away from a node representing the group G
  P_v = IKKBZ-solve-group($(V - (G \cup X), E), I, G, X \cup I$)
  mark all nodes in P_v
  for each $e \in E : (e = (U,W) \vee e = (W,U)) \wedge U \subseteq P_v \wedge \forall w \in W : w \notin X$
    if $\exists! u \in U : \neg$isMarked(u) add IKKBZ-precedence($Q, U, W, X + U$) as child of P_v
    else postpone e
  for each postponed edge $(e = (U,W) \vee e = (W,U)) \wedge \forall w \in W : w \notin X \wedge \forall u \in U$:isMarked(u)
    add IKKBZ-precedence($Q, U, W, X + U$) as child of P_v
  return P_v

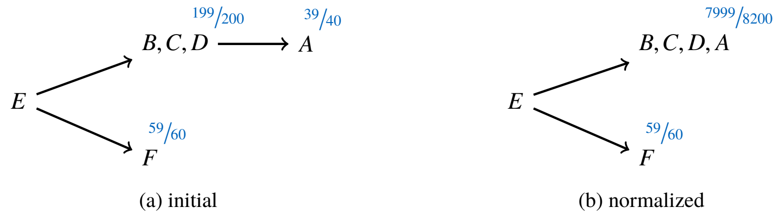
IKKBZ-normalize(c)
//normalize a chain of relations
  while $\exists i$: rank($c[i]$)>rank($c[i+1]$) $\wedge\neg$isPartial($c[i+1]$)
    merge $c[i]$ and $c[i+1]$ into a compound relation

building the precedence graph at the start relation E and directs all edges away from E. This is immediately possible for the edge (E,F). The forward hyperedge $(\{E\},\{C,D\})$ however, requires to solve the group $\{C,D\}$ first. This is done by recursively linearizing the precedence graphs for the subgraph covering $\{A,B,C,D\}$ rooted in C respectively D. Those precedence graphs, annotated with ranks and their respective linearizations are depicted in Fig 5. After cost comparison, $CBDA$ is selected as the best solution for the group, from which the algorithm picks the subchain CBD. The intermediate result of CBD has a cardinality of 200 which gives a rank of 199/200 for the group solution. Finally, the edge (B,A) is again regular and A is simply added as child of the compound node B,C,D. This completes the construction of the precedence graph which is depicted with annotated ranks in Fig. 4a.

Relations	Cardinality
A	100
B	100
C	50
D	50
E	100
F	120

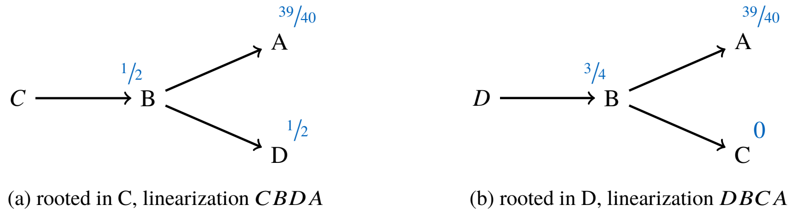
Join edge	Selectivity
(A,B)	0.4
(B,C)	0.02
(B,D)	0.04
({C,D},E)	0.01
(E,F)	0.5

Tab. 1: Cardinalities and selectivities for the example query (Fig. 1)



(a) initial (b) normalized

Fig. 4: The initial global precedence graph rooted in E for the example query in Fig. 1 and its normalization. Ranks are annotated in blue.



(a) rooted in C, linearization $CBDA$ (b) rooted in D, linearization $DBCA$

Fig. 5: Sub-precedence graphs when solving group $\{C, D\}$ rooted in C respectively D. Ranks are annotated in blue.

The second phase of the algorithm then builds a total order of relations based on this precedence graph. The algorithm descends into the tree until it encounters a node whose children are chains, which is immediately the case for the root node E. Before merging the child chains into a total order, any contradictory sequences UV within the chains are normalized if V is not partial. In the example, there is a contradictory sequence between the compound relation BCD and A. These two nodes are normalized into a compound node and the new rank computed accordingly (see Fig. 4b). After this normalization step all contradictions are resolved and the algorithm continues with merging the children of E. This results in the linearization $ECBDAF$. Based on this linearized search space, the polynomial time DP algorithm finally constructs the logical execution plan depicted in Fig. 6. Overall, the algorithm would build execution plans based on all linearizations for the different start relations and select the one with the lowest cost.

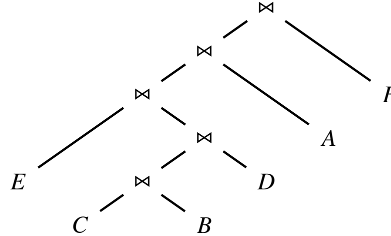


Fig. 6: Logical execution plan based on the linearization $ECBDAF$

Using this modified IK/KBZ algorithm we can now linearize the search space of arbitrary queries, including queries with non-inner joins and complex join predicates. If the query becomes too large for the DP step (for example more than 100 or 150 relations, depending on the available hardware) we can fall back to an iterative dynamic programming strategy using LinDP++ as inner algorithm, as discussed in [NR18].

4 Considering Potentially Beneficial Crossproducts

Having introduced a technique to handle non-inner joins in the previous section we now turn our attention onto the usefulness of crossproducts. Usually, when a join ordering algorithm does consider crossproducts, it considers all of them. Unfortunately this increases the search space dramatically, and increases the optimization time to $O(3^n)$, regardless of the structure of the query graph. And most of these considered crossproducts will be useless: A crossproduct $L \times R$ is inherently a $O(|L||R|)$ operation, while a hash join can be executed ideally in linear time. Which means that crossproducts are only attractive if at least one of its inputs is reasonably small. On the other hand crossproducts can sometimes be used to avoid repeated joins with large relations (by building the crossproduct of small relations first). We would like to capture this (rare, but useful) use case, without paying the exponential costs of considering all crossproducts. In this section we therefore introduce a cheap heuristic to detect potentially beneficial crossproducts. We use that information to make them explicit in the query graph: If a crossproduct between R_1 and R_2 is considered beneficial we add an artificial crossproduct edge with selectivity 1 between R_1 and R_2 to the query graph. The DP phase of LinDP++ will consider this edge during plan construction and will utilize the crossproduct if beneficial. Note that the heuristic itself is not tied to LinDP++, it could be used by any query graph based optimization algorithm, like, for example, DPhyp.

When finding crossproduct candidates we have two problems: First, finding good candidates has to be reasonably cheap, and second, we must be careful in the presence of non-inner joins, as adding crossproducts there can lead to wrong results. For example in the query $R_1 ⟗ (R_2 \bowtie R_3 \bowtie R_4)$ we must not introduce a crossproduct between R_1 and R_4, but we could introduce a crossproduct between R_2 and R_4. We solve that problem by analyzing

the paths between two relations: We only consider crossproducts between two relations R_i and R_j if there exists a path of regular (i. e., inner join) edges between them. This avoids bypassing non-inner joins with crossproducts.

Intuitively, a crossproduct is potentially beneficial if it allows to cheaply bypass a sequence of expensive join operations. Analyzing all paths between all pairs of relations is computationally expensive and not feasible in practice. However, restricting the analysis to paths of length two gives polynomial optimization time and still catches many cases where a crossproduct could result in a better plan: For a query Q we investigate all pairs of neighboring edges $e_1 = (u, v) \in Q$ and $e_2 = (v, w) \in Q$. We augment Q with an artificial crossproduct edge (v, w) of selectivity 1 if the cardinality of the crossproduct $|u \times w|$ is less than the result sizes of both of the joins:

$$|u \times w| < |u \bowtie v| \wedge |u \times w| < |v \bowtie w|$$

For the C_{out} cost function [CM95] this criterion gives all potentially beneficial crossproducts to bypass paths of length two while still keeping optimization complexity reasonable. Of course there could be even longer paths where bypassing joins via crossproducts would result in cheaper plans. Our experimental evaluation (Section 5), however suggests, that investigating paths of length two covers most of the important crossproducts, and has negligible overhead. If one wants to be more aggressive with crossproducts we could consider even longer paths if the relations are particular small (for example a single tuple). But this leads to diminishing return compared to the optimization time, which is why we used only paths of length two in our experiments.

Note that crossproducts should be introduced very conservatively, especially if cardinality estimates are inaccurate. A crossproduct is inherently quadratic in nature, and if an input relation has estimated cardinality of 1 and a real cardinality of 10,000 (which can easily happen in practice), the performance impact will be disastrous. On the other hand the cardinality is sometimes known exactly, for example if the primary key is bound or if the input is the result of a group-by query with known group count, which makes the introduction safe. For base tables the available statistics can sometimes provide reasonably tight upper bounds for the input size, which also makes the computation safe if the upper bound is used in the formula above. This essential prudence reduces the number of cases where we will consider crossproducts, but nevertheless there remain queries where crossproducts are attractive and safe, and we can and should consider them during join tree construction.

Consider for example the query graph with cardinalities and selectivities given in Fig. 7a. The optimal execution plan without crossproducts for this query has costs of 1.84M and is depicted in Fig. 8a. When applying the crossproduct heuristic, the query graph is augmented with two additional edges (A, C) and (D, F) as shown in Fig. 7b. While these two additional edges only marginally enlarge the search space, costs are cut by almost 50% to 0.94M using the plan shown in Fig. 8b. Note that even considering all possible crossproducts, although a much larger search space is explored, does not uncover a cheaper plan. Further note that LinDP++ generates the same plan, despite exploring a reduced, linearized search space.

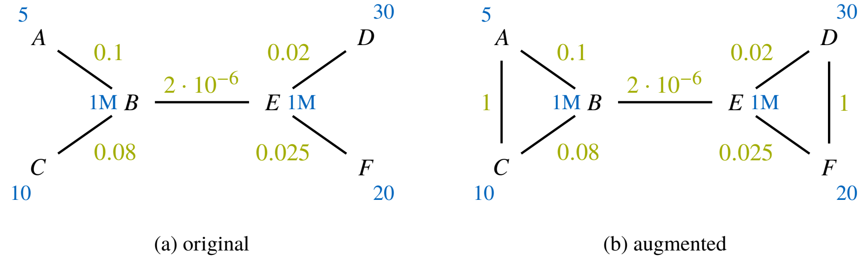


Fig. 7: Query graph where crossproducts enable cheaper execution plans. Cardinalities are annotated in blue, join selectivities in green.

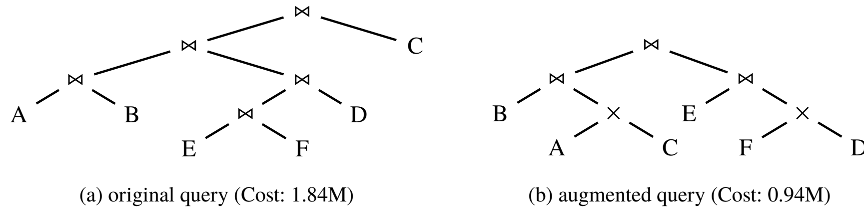


Fig. 8: Optimal execution plans for the query graphs depicted in Fig. 7

5 Evaluation

In this section we present the results of an extensive experimental analysis of the techniques described in this paper. We compare LinDP++ against a multitude of different join ordering algorithms including DPhyp [MN08], Greedy Operator Ordering (GOO) [Fe98], Iterative Dynamic Programming [KS00] using DPhyp as inner algorithm, Quickpick [WP00], genetic algorithms [SMK97], query simplification [Ne09], minsel [Sw89], and linearized DP [NR18]. The algorithms were used to optimize the queries of the following well known standard benchmarks using the C_{out} [CM95] cost function: TPC-H [Tra17b] and TPC-DS [Tra17a], LDBC BI [An14], the Join Order Benchmark (JOB) [Le18], and the SQLite test suite [Hi15].

The query graphs of the standard benchmark queries unfortunately contain only a small number of hyperedges, most of them do not contain any hyperedges at all. Furthermore, the queries are fairly small and can all easily be optimized by a full hypergraph based DP algorithm [NR18]. Nevertheless, large queries with outer joins are a reality we have to deal with [Vo18]. To thoroughly assess our approach also for larger queries with non-inner joins we thus additionally evaluate LinDP++ on a synthetic workload of large randomly generated queries. We use the same set of tree queries used in [NR18] which was generated using the procedure described in [Ne09]. The set contains 100 different random queries per size class. Sizes range from 10 to 100 relations per query, which gives a total of 1,000 queries. Hyperedges are introduced to the queries by randomly adding artificial reordering

Algorithm	TPC-H	TPC-DS	LDBC	JOB	SQLite
DPhyp	0.4	90	1.2	227	2.2K
GOO	0.8	9.5	2.2	13.7	61
linearized DP	1.4	18.7	4.4	33.4	4.7K
LinDP++	1.6	19.9	4.4	36.2	4.7K

Tab. 2: Total optimization time (ms) for standard benchmarks

constraints between neighboring joins. All algorithms were ran single-threaded with a timeout of 60 seconds on a 4 socket Intel Xeon E7-4870 v2 at a clock rate of 2.3 GHz with 1 TB of main memory attached. When comparing the quality of the plans for a query generated by different algorithms we report *normalized costs*, i. e. the factor by which a plan is more expensive than the best plan found by any of the algorithms.

5.1 Hypergraph Handling

The algorithm described in this paper targets query sizes far beyond what exact algorithms with exponential runtime can solve in a reasonable amount of time. Thus we start by analyzing the runtime characteristics of LinDP++.

For completeness reasons we first report numbers for all considered standard benchmarks even though their queries are all rather small and a full graph based DP would be the algorithm of choice here. In Tab. 2 we summarize optimization time of DPhyp, GOO, linearizedDP, and LinDP++ for all considered benchmarks. Most of the queries are optimized almost instantly and optimization times of LinDP++ are comparable to those of linearized DP. The only benchmark where optimization time becomes noticeable is the SQLite test suite, which contains more than 700 queries on up to 64 relations. But even the largest query of the SQLite test suite is optimized by LinDP++ in 27ms. Regarding the quality of the plans generated by LinDP++: 93% of the plans are indeed optimal, 6% of them are suboptimal by at most a factor of 2 and only 10 of the 1159 execution plans are worse than that. Note: we compare plans to the best plan found when considering all valid crossproducts here.

Once queries become more complex and exact optimization becomes infeasible, search space linearization helps to keep optimization times reasonable. With LinDP++ we are now able to linearize the search space of queries with non-inner joins, a class of queries that could not be handled by linearized DP. To see whether this ability to linearize hypergraph queries comes at the expense of optimization performance we compare linearized DP on regular queries with LinDP++ on hypergraph queries with the same number of relations. Figure 9 shows median, minimum and maximum optimization time per size class for linearized DP and LinDP++. The overhead incurred by hypergraph handling is negligible and LinDP++ is just as well able to optimize queries on 100 relations within 100ms on average.

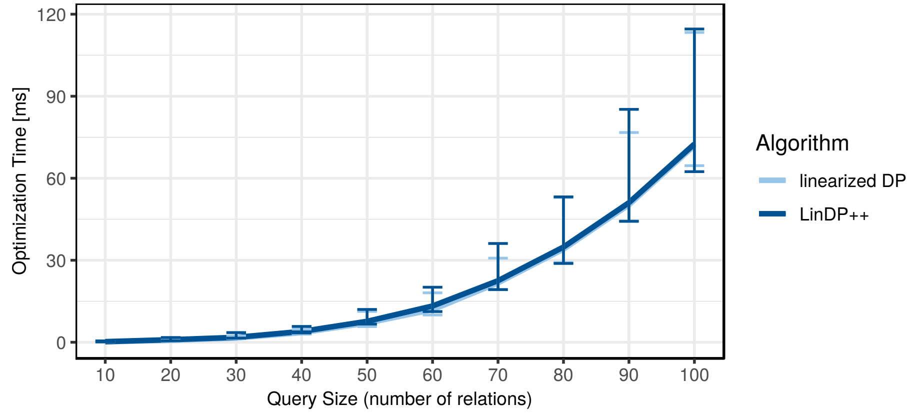


Fig. 9: Median optimization times for LinDP++ on hypergraph queries compared to linearized DP on regular graph queries for queries on up to 100 relations. The error bars indicate minimum and maximum optimization time per size class.

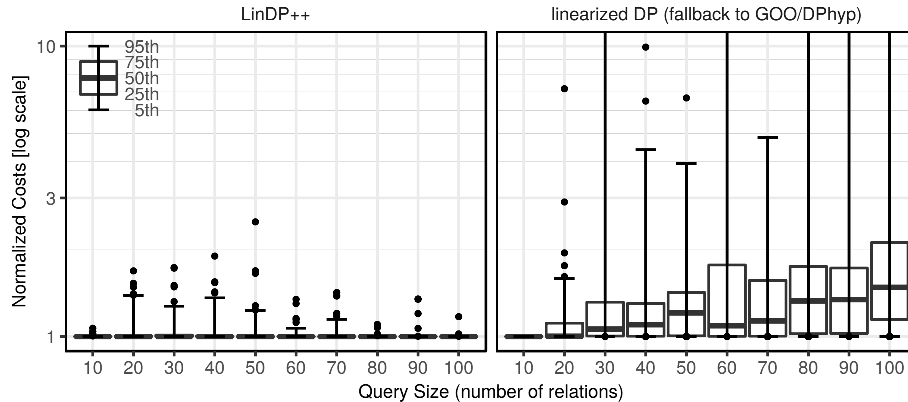


Fig. 10: Distribution of normalized costs of LinDP++ plans compared to the plans the hypergraph aware greedy GOO/DPhyp fallback of linearized DP generates for queries on up to 100 relations.

The original adaptive optimization framework [NR18] had to fall back to the greedy iterative DP with DPhyp as hypergraph aware inner algorithm (GOO/DPhyp) for large queries with non-inner joins. Depending on the structure of the query graph, this could already be the case for hypergraph queries touching as few as 14 relations. Using the generalized LinDP++ technique we can avoid the greediness for these queries and generate significantly better plans. Figure 10 compares the normalized costs of the plans generated by LinDP++ with the ones GOO/DPhyp generates for the synthetic workload with hyperedges. On average, plan costs are within 2% of the best plan and 814 plans are indeed the best known plans for their respective query and normalized costs of 123 plans are within 10% of

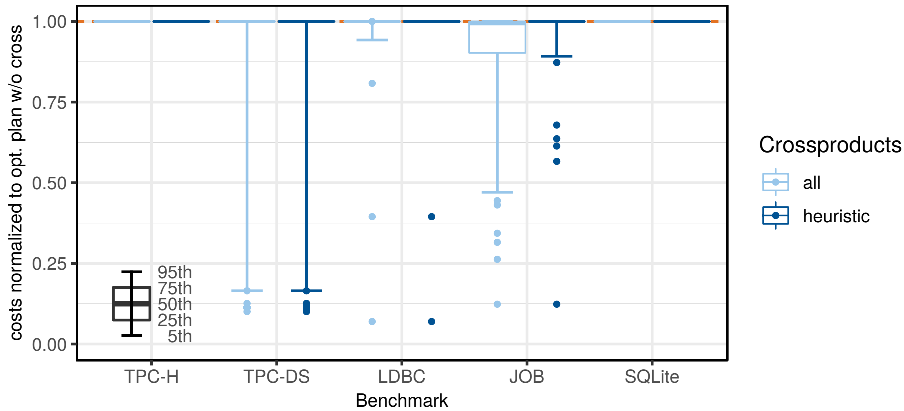


Fig. 11: Normalized costs of plans when either all valid crossproducts or some explicit crossproduct edges as suggested by the heuristic are considered. Costs are normalized to the optimal plan without crossproducts.

the best. The remaining execution plans have normalized costs below 2 with the exception of one query, for which the plan is 2.4 times as expensive as the best known plan. In contrast to that, 139 plans generated by GOO/DPhyp already have normalized cost worse than 2 and costs of 54 plans are worse than the best plan by a factor of 10 or more.

5.2 Crossproduct Benefits

Investigating the effectiveness of the crossproduct heuristic described in Section 4 on the queries of the standard benchmarks shows that crossproducts can indeed improve execution plans. Figure 11 summarizes the costs of plans normalized to the optimal plan without crossproducts. We compare these normalized costs when considering all crossproducts with those when considering only the crossproducts suggested by our heuristic. On average, introducing crossproduct edges improves plan cost by up to 18%, depending on the benchmark. Nevertheless, 90% of the execution plans remain the same even when considering all valid crossproducts. This confirms our statement, that the vast majority of possible crossproducts is irrelevant and should not be considered. However, while plan improvements are minimal for most queries, a maximum cost reduction of a factor of 14.4 reconfirms the claim of Ono and Lohman [OL90], that some crossproducts can significantly improve plan quality. The figure also shows, that our simple heuristic indeed already covers many of the relevant crossproducts. Only the Join Order Benchmark would benefit from additional crossproducts that bypass larger chains of joins (mostly of length 3). Extensively investigating all crossproduct possibilities during join ordering is thus neither required to get good plans nor feasible in terms of optimization complexity.

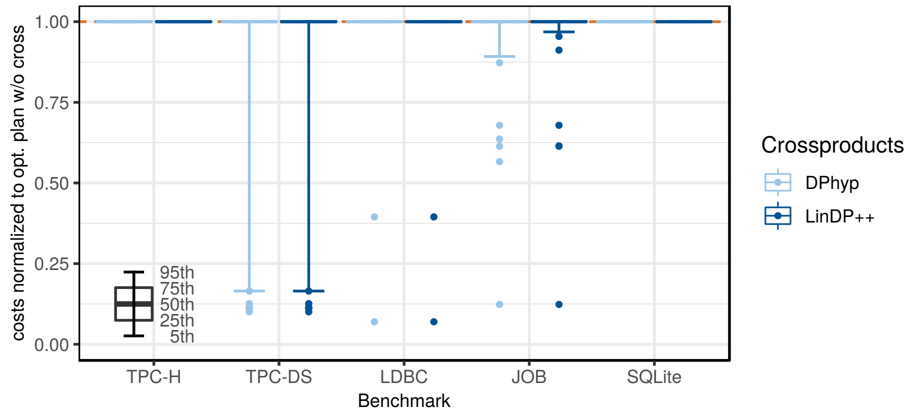


Fig. 12: Normalized costs of plans generated by LinDP++ compared to the optimal plans with some crossproducts as suggested by the heuristic. Costs are normalized to the optimal plan without crossproducts.

crossproducts are never considered during the linearization phase of LinDP++, as they are eliminated when removing cycles from the query graph. However, even though they are ignored by the first phase, the second phase does consider them and plan costs are reduced almost as much as with a full DP algorithm. Figure 12 shows the differences in cost improvements comparing LinDP++ against full DPhyp, both operating on the augmented query graph. Despite the reduced search space, which gives much better optimization times, most of the beneficial crossproducts are discovered and plan costs are within 1% of the DPhyp solutions on average.

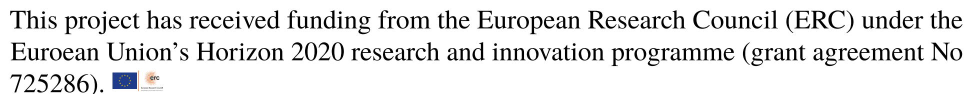
6 Conclusion

In this paper we eliminate a severe limitation of the recently proposed adaptive join ordering framework [NR18]. While generating high quality execution plans for many large queries using search space linearization, the framework had to fall back to a greedy heuristic as soon as a large query contained a single outer join. The generalized algorithm LinDP++ described in this paper enables linearization of the search space of arbitrary queries, including those with non-inner joins. We experimentally show that the join orders generated by LinDP++ are clearly superior to those generated by the greedy fallback of the original framework.

LinDP++ in addition is equiped with a fast heuristic to detect promising opportunities to perform a crossproduct. Despite considering some crossproducts, LinDP++ deliberately avoids looking at all crossproducts which would result in exponential search space size. Furthermore, the heuristic ensures that any considered crossproduct obeys all reordering constraints induced by non-inner joins. We demonstrate the effectiveness of this heuristic on the queries of major database benchmarks. The heuristic detects most relevant crossproduct

opportunities while keeping the search space small and thus optimization time reasonable. Our experiments further verify that considering all valid crossproducts is not worth the dramatically increased optimization time, as the additionally considered crossproducts rarely lead to an additional cost reduction.

Even a polynomial time heuristic like LinDP++ becomes too expensive at some point. At that scale, iterative dynamic programming has proven to be an effective technique, as it allows to gracefully tune down plan quality in favor of acceptable optimization times. According to our experiments however, the chosen greedy algorithm – Greedy Operator Ordering (GOO) – seems to also have quality issues with non-inner joins. To ensure high quality execution plans even at that scale we thus would like to investigate alternatives or improve GOO in this setting.

This project has received funding from the European Research Council (ERC) under the Euroean Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

References

[An14] Angles, Renzo; Boncz, Peter A.; Larriba-Pey, Josep-Lluis; Fundulaki, Irini; Neumann, Thomas; Erling, Orri; Neubauer, Peter; Martínez-Bazan, Norbert; Kotsev, Venelin; Toma, Ioan: The linked data benchmark council: a graph and RDF industry benchmarking effort. SIGMOD Record, 43(1):27–31, 2014.

[CM95] Cluet, Sophie; Moerkotte, Guido: On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In: Proceedings of ICDT ’95. pp. 54–67, 1995.

[Fe98] Fegaras, Leonidas: A New Heuristic for Optimizing Large Queries. In: Proceedings of DEXA ’98. pp. 726–735, 1998.

[FM13] Fender, Pit; Moerkotte, Guido: Top down plan generation: From theory to practice. In: Proceedings of ICDE 2013. pp. 1105–1116, 2013.

[GLP93] Gallo, Giorgio; Longo, Giustino; Pallottino, Stefano: Directed Hypergraphs and Applications. Discrete Applied Mathematics, 42(2):177–201, 1993.

[Gr95] Graefe, Goetz: The Cascades Framework for Query Optimization. IEEE Data Eng. Bull., 18(3):19–29, 1995.

[Ha08] Han, Wook-Shin; Kwak, Wooseong; Lee, Jinsoo; Lohman, Guy M.; Markl, Volker: Parallelizing query optimization. PVLDB, 1(1):188–200, 2008.

[Hi15] Hipp, R. et al.: , SQLite (Version 3.8.10.2). SQLite Development Team. Available from `https://www.sqlite.org/download.html`, 2015.

[IK84] Ibaraki, Toshihide; Kameda, Tiko: On the Optimal Nesting Order for Computing N-Relational Joins. ACM Trans. Database Syst., 9(3):482–502, 1984.

[KBZ86] Krishnamurthy, Ravi; Boral, Haran; Zaniolo, Carlo: Optimization of Nonrecursive Queries. In: Proceedings of VLDB ’86. pp. 128–137, 1986.

[KS00] Kossmann, Donald; Stocker, Konrad: Iterative dynamic programming: a new class of query optimization algorithms. ACM Trans. Database Syst., 25(1):43–82, 2000.

[Le18] Leis, Viktor; Radke, Bernhard; Gubichev, Andrey; Mirchev, Atanas; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: Query optimization through the looking glass, and what we found running the Join Order Benchmark. VLDB J., 27(5):643–668, 2018.

[MFE13] Moerkotte, Guido; Fender, Pit; Eich, Marius: On the correct and complete enumeration of the core search space. In: Proceedings of SIGMOD 2013. pp. 493–504, 2013.

[MN06] Moerkotte, Guido; Neumann, Thomas: Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In: Proceedings VLDB 2006. pp. 930–941, 2006.

[MN08] Moerkotte, Guido; Neumann, Thomas: Dynamic programming strikes back. In: Proceedings of SIGMOD 2008. pp. 539–552, 2008.

[MS79] Monma, Clyde L.; Sidney, Jeffrey B.: Sequencing with Series-Parallel Precedence Constraints. Math. Oper. Res., 4(3):215–224, 1979.

[Ne09] Neumann, Thomas: Query simplification: graceful degradation for join-order optimization. In: Proceedings of SIGMOD 2009. pp. 403–414, 2009.

[NR18] Neumann, Thomas; Radke, Bernhard: Adaptive Optimization of Very Large Join Queries. In: Proceedings of SIGMOD 2018. pp. 677–692, 2018.

[OL90] Ono, Kiyoshi; Lohman, Guy M.: Measuring the Complexity of Join Enumeration in Query Optimization. In: Proceedings of VLDB 1990. pp. 314–325, 1990.

[Se79] Selinger, Patricia G.; Astrahan, Morton M.; Chamberlin, Donald D.; Lorie, Raymond A.; Price, Thomas G.: Access Path Selection in a Relational Database Management System. In: Proceedings of SIGMOD 1979. pp. 23–34, 1979.

[SMK97] Steinbrunn, Michael; Moerkotte, Guido; Kemper, Alfons: Heuristic and Randomized Optimization for the Join Ordering Problem. VLDB J., 6(3):191–208, 1997.

[Sw89] Swami, Arun N.: Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In: Proceedings of SIGMOD 1989. pp. 367–376, 1989.

[TK17] Trummer, Immanuel; Koch, Christoph: Solving the Join Ordering Problem via Mixed Integer Linear Programming. In: Proceedings of SIGMOD 2017. pp. 1025–1040, 2017.

[Tra17a] Transaction Processing Performance Council. TPC Benchmark DS, 2017.

[Tra17b] Transaction Processing Performance Council. TPC Benchmark H, 2017.

[VM96] Vance, Bennet; Maier, David: Rapid Bushy Join-order Optimization with Cartesian Products. In: Proceedings of SIGMOD 1996. pp. 35–46, 1996.

[Vo18] Vogelsgesang, Adrian; Haubenschild, Michael; Finis, Jan; Kemper, Alfons; Leis, Viktor; Mühlbauer, Tobias; Neumann, Thomas; Then, Manuel: Get Real: How Benchmarks Fail to Represent the Real World. In: Proceedings of DBTest@SIGMOD 2018. pp. 1:1–1:6, 2018.

[WC18] Wang, TaiNing; Chan, Chee-Yong: Improving Join Reorderability with Compensation Operators. In: Proceedings of SIGMOD 2018. pp. 693–708, 2018.

[WP00] Waas, Florian; Pellenkoft, Arjan: Join Order Selection - Good Enough Is Easy. In: Proceedings of the 17th BNCOD. pp. 51–67, 2000.