

Adaptive Optimization of Very Large Join Queries

Thomas Neumann

Technische Universität München
neumann@in.tum.de

Bernhard Radke

Technische Universität München
radke@in.tum.de

ABSTRACT

The use of business intelligence tools and other means to generate queries has led to great variety in the size of join queries. While most queries are reasonably small, join queries with up to a hundred relations are not that exotic anymore, and the distribution of query sizes has an incredible long tail. The largest real-world query that we are aware of accesses more than 4,000 relations. This large spread makes query optimization very challenging. Join ordering is known to be NP-hard, which means that we cannot hope to solve such large problems exactly. On the other hand most queries are much smaller, and there is no reason to sacrifice optimality there.

This paper introduces an adaptive optimization framework that is able to solve most common join queries exactly, while simultaneously scaling to queries with thousands of joins. A key component there is a novel *search space linearization* technique that leads to near-optimal execution plans for large classes of queries. In addition, we describe implementation techniques that are necessary to scale join ordering algorithms to these extremely large queries. Extensive experiments with over 10 different approaches show that the new adaptive approach proposed here performs excellent over a huge spectrum of query sizes, and produces optimal or near-optimal solutions for most common queries.

CCS CONCEPTS

• Information systems → Query optimization;

ACM Reference format:

Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of 2018 International Conference on Management of Data, Houston, TX, USA, June 10–15, 2018 (SIGMOD'18)*, 16 pages.
<https://doi.org/10.1145/3183713.3183733>

1 INTRODUCTION

Joins are the backbone of query processing. They occur in nearly every query, and they can affect query runtime dramatically. Choosing a proper join order is thus one of the most important, if not *the* most important task of the query optimizer. Besides joins being ubiquitous, the huge variety in join queries adds to the complexity of the problem. Most join queries are reasonably small, joining less than 20 relations. But the advent of business intelligence tools has

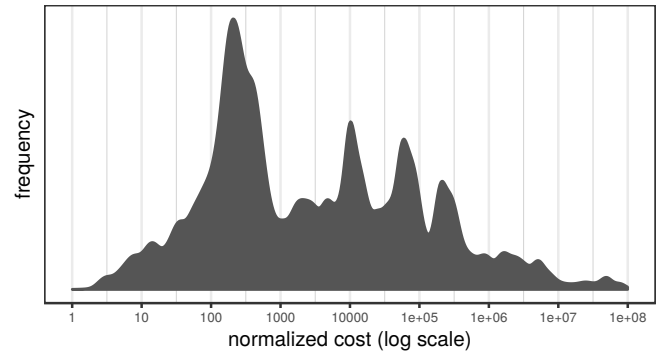


Figure 1: Normalized Cost Distribution of Random Plans for a Data-Warehouse-Style Query with 50 Relations

lead to (generated) ad-hoc queries that can easily touch a hundred relations, and a database system must be able to handle these, too.

Even moderately sized queries with, e.g., 50 relations are far beyond what can be optimized exactly. In such cases, optimizers have to sacrifice optimality and employ heuristics to keep optimization time reasonable. Figure 1 shows the distribution of the costs normalized to the best plan of 10,000 random plans for a data-warehouse style query with 50 relations. The cost of most plans are at least 100× higher than the cheapest plan found. At that scale, it becomes hard for a heuristic to find one of the very few good plans.

This however is by far not the end of the spectrum, there is an incredible long tail in query sizes. The largest real-world query that we are aware of includes, after view expansion, 4,598 relations ([19]). Admittedly these mega queries [7] are outliers even in the SAP context (the next largest has 2,298 relations and only a handful have over 1,000), but queries with several hundred relations are not that uncommon in this workload. The public query log from the Tableau Public data visualization tool shows a similar distribution, with most queries being small, but a long tail of queries with more than 100 relations reaching up to 369 in a single join query. Note that especially in such exploratory scenarios, the workload is not known up-front and most queries are issued only once. Other techniques to improve query performance or reduce their complexity like, e.g., materialization, may thus not always be desirable or applicable.

Large ad-hoc queries thus are a reality that database systems have to deal with. PostgreSQL for example uses dynamic programming to find the optimal join order for queries with less than 12 relations and switches to genetic algorithms for larger queries. DB2 uses dynamic programming and switches to a greedy strategy when the query becomes too large. Other systems use similar fallbacks. Often these switching points imply “falling off a cliff”, i.e., we get good plans up to a certain point, and significantly worse results once queries get slightly larger, which is highly unsatisfying. The query optimizer should try to solve the problem exactly, and adaptively tune down the result quality if optimality can no longer be guaranteed.

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

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-4703-7/18/06...\$15.00

<https://doi.org/10.1145/3183713.3183733>

Admittedly, the huge spread in complexity causes immense problems for the query optimizer. Fundamentally, the problem is known to be NP-hard [13], which would seem to suggest that we cannot do better than use a heuristic anyway for all but the smallest queries. But this argument is defeatist. In reality, we can optimize surprisingly large queries exactly, and even if we cannot, we can find very good join orders for even larger classes of queries.

In this paper we therefore introduce an adaptive optimization framework that finds optimal solutions for most common queries, near optimal solutions for very large classes of queries, and that scales down gracefully for mega queries with up to 5,000 relations. We achieve this by combining dynamic programming with a novel search space linearization technique, and by introducing implementation tricks to existing algorithms that are necessary to handle very large queries efficiently. Extensive experiments show that this combination works extremely well, and allows us to build a query optimizer that can handle the whole query spectrum efficiently.

Looking at the immense span of query complexity from 2 to 5,000 relations, one has to realize that there are different requirements and expectations depending on the query size. For small queries, which make up the bulk of most workloads, we clearly want to find the optimal order. For medium sized queries of up to 100 relations, which are still quite common, we in general can no longer guarantee optimality, but we want to be close to optimal. For large queries of up to 1,000 relations, which are rare, we must accept that we cannot find the best plan, but we nevertheless want good results, and we want quality to degrade gracefully. For the unique mega-queries with more than 1,000 relations we must be happy if we are able to construct a decent plan at all; most optimizers simply break for such queries [7]. Our adaptive framework handles this span gracefully, combining good or even optimal plans with low optimization times across the whole spectrum from 2 to 5,000 relations.

The rest of this paper is structured as follows: Section 2 formalizes the problem, and Section 3 discusses related work. Then, we introduce our adaptive optimization framework in Section 4. The various implementation details necessary for very large queries are shown in Section 5. The algorithms are evaluated in Section 6. Finally, we draw conclusions and discuss future work in Section 7.

2 SETTING

Before going into algorithms, we briefly formalize the problem. This paper is targeted at query optimizers that can be used in commercial database systems, therefore we have to support all kinds of SQL queries, including unusual predicates and non-inner joins. We assume that we are given the query in the form of a *query graph*, as shown in Figure 2: A query graph $G = (V, E)$ has as nodes V the set of relations, and as edges E the join possibilities as implied by the join conditions of the query. We say that a join tree T *adheres* to a query graph G if for every subtree $T' = T_1 \bowtie T_2$ of T there exist relations R_1, R_2 such that $R_1 \in T_1$ and $R_2 \in T_2$ and $(R_1, R_2) \in E$. Now given a query graph G and a cost function C , the task is to construct a join tree T that adheres to G and that minimizes C .

Note that in the general case, namely for queries with non-inner joins, the query graph can be a hyper-graph instead of a graph [22]. This means that a join edge connects not just two relations but instead sets of relations, and accordingly, R_1 and R_2 in our adherence

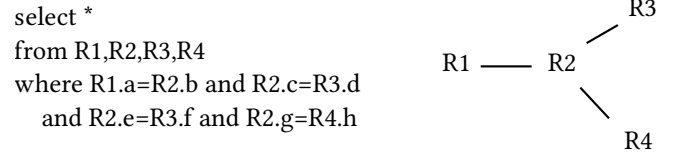


Figure 2: A Query and its Query Graph

definition can be sets of relations instead of single relations. We assume that all reordering constraints for non-inner joins have been encoded in the query graph, as described in [20].

For the algorithms we assume that the query graph is connected. If the original query graph is not connected (i.e., if it contains implicit cross products), we connect the connected components with cross product edges. Note that we do not consider (additional) implicit cross products during optimization. There are two reasons for that. First, there is the performance argument, as ignoring cross products dramatically reduces the search space, and in practice cross products are rarely useful: A cross product is inherently an $O(n^2)$ operation, which means that it only makes sense for very small relations. This observation is the reason why most existing systems ignore cross products, too. Even more important however is a correctness argument: In the presence of non-inner joins, introducing cross products between arbitrary relations can lead to wrong results. We therefore always adhere to the query graph. If a system deems cross products attractive for a particular query, for example because there are two very small relations involved, it must explicitly add that join possibility to the query graph, updating the edges for non-inner joins as needed.

For the cost function the dynamic programming parts assume nothing except the Bellman principle, any reasonable cost function will do. The ranking steps used by some algorithms require a cost function with ASI property (as discussed in Section 3), for example the C_{out} function that minimizes the size of intermediate results. If the overall cost function does not have ASI properties we can still use C_{out} for ranking and the true cost function for the dynamic programming part. As shown in [17], simple cost functions like C_{out} [5] or C_{mm} [17] are usually good enough even for full plans.

We describe all algorithms as constructive ones, i.e., we construct a join tree for a given query graph. When integrating them into a transformative optimizer, the technique from [25] can be used.

3 RELATED WORK

Selinger et al. pioneered the use of dynamic programming (DP) for join ordering by constructing optimal partial plans of increasing size [24]. We will refer to that algorithm as *DPSize*. To reduce the search space, many systems only consider left-deep instead of bushy trees. We call that variant *DPSizeLinear*. Considering left-deep trees only is tempting, as this significantly speeds up optimization, but bushy plans are often much more efficient than linear plans.

This size-based dynamic programming is well known and widely used, but it is not the most efficient approach. Graph-based dynamic programming strategies that organize the search by the structure of the query graph can perform significantly better than *DPSize*, as they avoid considering relation combinations that cannot lead to a final join tree [21]. There have been a number of improvements to that original approach, including top-down formulations that

can outperform bottom-up DP algorithms for some queries as they allow for cost-based pruning [6, 9] and generalizations to non-inner joins [22]. We will refer to the latter approach as *DPHyp*.

While the join ordering problem is NP-hard in general, this does not mean that every instance of the problem is hard. If the query graph forms a chain, *DPHyp* can find the optimal solution in $O(n^3)$, which is tractable even for relatively large values of n^1 . Other query shapes are more difficult. As the difficulty follows from the shape of the query graph, we can 1) predict the optimization time by looking at the query graph, and 2) reduce the optimization time by manipulating the query graph. This observation led to the idea of query *simplification*, where the query graph is successively made more restrictive using a greedy heuristic until it becomes tractable for dynamic programming [23]. This works very well if the query is just a bit too complex for dynamic programming, as then only a few greedy steps are needed, but for very large queries this approach breaks as then the greedy step dominates and results can be poor.

Another idea to handle dynamic programming for large queries is Iterative Dynamic Programming [14]. It combines dynamic programming with a greedy step, and comes in two flavors: The IDP-1 algorithm from [14] runs *DPSize* up to a given size k , greedily chooses the cheapest plan of that size, conceptually transforms it into a base relation, and repeats the processes until all relations have been joined. While plausible in general, the approach has the problem that each iteration has a runtime in $O(n^k)$, which means that for large values of n we have to choose very small values for k , which lets the algorithm degenerate into a greedy approach. More relevant for our use case is the IDP-2 variant, which first constructs a complete join tree using a greedy approach, and then optimizes subtrees of size k using DP. This works well even for large queries, and we refer to that algorithm as *iterative dp*.

Greedy algorithms are commonly used to optimize large queries. One commonly used variant is the *min-sel* heuristic, where the joins are ordered by increasing selectivity [27]. There are multiple ways to implement it, the most elaborate one tries out every relation as start relation, and tries the most selective sequence for each start relation, picking the cheapest overall. While easy to implement, the quality of the generated plans can be quite poor. And due to the repeated computations the algorithm is not as cheap as one might think, which becomes noticeable for extremely large queries. One of the best known greedy heuristics is Greedy Operator Ordering [8], which greedily constructs bushy join trees by repeatedly picking the pair with the minimum result cardinality. The result quality is usually good, and the algorithm can handle large queries as well, at least when implemented carefully. We refer to it as *GOO* within the paper. Recent follow-up work on *GOO* explores alternative plans by considering relation pull-up or push-down during tree merge [2]. We do not consider this variant here, as the additional exploration adds non-negligible optimization overhead. Other approaches use meta-heuristics or randomized algorithms to find good join orders [26], but the result quality can be quite poor.

Besides algorithms specifically designed for query optimization, there has been work on utilizing generic solvers for join ordering [30]. By translating the query graph into a mixed integer linear

program (MILP) existing solvers can be used to obtain a linear join tree. Such an approach benefits from the decades of research effort put into those solvers. Furthermore, such solvers can be stopped at any time and still deliver a solution (although no longer optimal). However, this approach considers cross products which makes the search space much larger. And the complexity of the join ordering problem remains, whether it is solved by specialized algorithms or by a general purpose solver. Hence, utilizing MILP solvers is not an alternative for the large queries considered in this paper.

Another very interesting approach is the *IKKBZ* family of algorithms [13, 15], called *IKKBZ* in this paper. It can construct the optimal join plan in polynomial time, which is an extremely attractive property, but it comes with a number of limitations. First, it needs a cost function that has *Adjacent Sequence Interchange (ASI)* properties, which means that it must be possible to compute a cost/benefit ratio (called the rank) for every join. That is possible for cost functions like *C_{out}*, but not in general. Second, it requires an acyclic query graph. And finally, it can only construct linear join trees, which are inferior to bushy trees. These limitations usually prevent practical usage, but its good properties are so attractive that we will use it as intermediate step within this paper.

One of the few papers that explicitly discusses large queries is [7]. They show that existing optimizers are often simply unable to handle queries with 1,000 relations, in particular the transformative approaches used in many commercial systems. Instead of optimizing these mega queries as a whole, the authors propose to greedily select parts of the query and to optimize only those. Another paper also mentions queries with thousands of tables [4], but few details are given beyond constructing partial join orders greedily.

4 ADAPTIVE OPTIMIZATION

After discussing related work, we now introduce our adaptive optimization framework. It analyzes the query graph, and then picks the most appropriate algorithm: If the query is small enough (or more precisely: if the query graph is simple enough), it uses dynamic programming to construct the optimal join tree. If that is not possible within the given optimization budget, it switches to a heuristic. But instead of switching to a greedy approach, as some other systems do, it uses a novel search space linearization technique to make dynamic programming tractable. This works extremely well in practice (see Section 6) and allows us to construct optimal or near-optimal solutions for queries with up to 100 relations. For even larger queries we have to introduce a greedy step, but we again use the linearization to improve the greedy solution. We now discuss the different optimization stages in more detail.

4.1 Small Queries

For small queries we clearly want to find the optimal join order. The only question is: what is a small query? The answer to this question depends on the shape of the query graph: For the best case, chain queries with a linear query graph, we can easily solve queries with 100 relations exactly with a graph-based DP like *DPHyp*, as the algorithms complexity is $O(n^3)$ then. For the worst case, clique queries, however the runtime is in $O(3^n)$, which means we can probably only solve queries of with about 14 relations exactly. Perhaps slightly more than 14 on fast machines, but not much more,

¹*DPSize* needs $O(n^4)$ for chain queries, which is still polynomial, but stops being tractable for much smaller values of n .

```

countCC( $Q = (V, E)$ , budget)
// counts the number of connected subgraphs of  $Q$ 
// stops after finding budget graphs
  label node  $v$  in  $V$  from 0 to  $|V|-1$ 
   $c=0$ 
  for each  $v_i \in V$ 
    if ( $c = c + 1$ ) > budget return  $c$ 
     $B_i = \{v_j | v_j \in V \wedge j < i\}$ 
     $c = \text{countCCRec}(Q, v_i, B_i, c, \text{budget})$ 
  return  $c$ 

countCCRec( $Q = (V, E), S, X, c, \text{budget}$ )
// expands the subgraph  $S$  of  $Q \setminus X$  recursively
// updates the count  $c$  for every subgraph
   $N_S = \{v | v \in V \wedge v \notin X \wedge \exists s \in S \wedge s \text{ is connected to } v\}$ 
  for each  $S' \subseteq N_S$ 
    if ( $c = c + 1$ ) > budget return  $c$ 
     $c = \text{countCCRec}(Q, S \cup S', X \cup N_S, c, \text{budget})$ 
  return  $c$ 

```

Figure 3: Counting the number of connected subgraphs of a query graph Q with a given enumeration budget

the exponential growth kills the algorithm. Real queries are in between these extremes, and it is not obvious how to predict the optimization time.

However, what we can do reasonably cheaply is counting the number of connected subgraphs of a query graph. The number of connected subgraphs is identical to the size of the full dynamic programming table [21], i.e., the memory consumption of the DP algorithm, and indirectly determines its optimization time. If the number of connected subgraphs is reasonably small, for example up to 10,000, we know that a graph-based DP algorithm will be fast. This is the case for the examples given above, i.e., chain queries with up to 100 relations and clique queries with less than 14 relations.

Thus, if the query joins less than 14 relations, we unconditionally use DPHyp [22] to find the optimal join order. Otherwise, if the query has up to 100 relations, we use the algorithm from Figure 3 to count the number of connected subgraphs up to the budget of 10,000. It counts the subgraphs by choosing each relation as start node, and then recursively expands the subgraph, skipping the parts of the graph that will be handled by functions higher up in the call stack. The algorithm is a streamlined variant of DPHyp that is reduced to the graph traversal without any DP table. It is much faster than a real DP algorithm because we are only interested in the count, not in the actual join trees, and it stops once we have seen more than the given budget because we only want to know if the count is within the budget or not. If the number of connected subgraphs is within our chosen budget of 10,000, we solve the query optimally using DPHyp, regardless of the number of relations. Otherwise, if dynamic programming is no longer feasible, we switch to the search space linearization stage that we will describe next.

4.2 Medium Queries

At some point we simply cannot use DP any more, because optimization becomes too expensive. However, we have seen that the

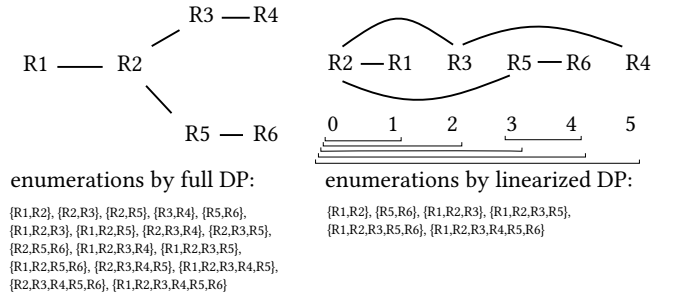


Figure 4: Example for Search Space Linearization

```

IKKBZ( $Q = (V, E)$ )
// construct an optimal left-deep tree
// for the acyclic query graph  $Q$ 
   $b = \emptyset$ 
  for each  $v \in V$ 
     $P_v = Q$  directed away from  $v$ 
    while  $P_v$  is not a chain
      pick  $v'$  in  $P_v$  that has chains as input
      IKKBZ-normalize each input chain of  $v'$ 
      merge the input chains by rank
    if  $b = \emptyset \vee C(P_v) < C(b)$ 
       $b = P_v$ 
  return  $b$ 

IKKBZ-normalize( $c$ )
// normalizes the chain  $c$  such that it is sorted by rank
  while  $\exists i: \text{rank}(c[i]) > \text{rank}(c[i+1])$ 
    merge  $c[i]$  and  $c[i+1]$  into a compound relation

```

Figure 5: The IKKBZ algorithm [13, 15]

moment where DP becomes too expensive depends upon the shape of the query graph. For linear query graphs we can solve quite large queries exactly, while cliques or stars are much more difficult to optimize. Therefore, we make medium sized queries with up to 100 relations tractable for dynamic programming by *linearizing* the search space. The core idea of search space linearization is that we restrict the DP algorithm to consider only connected subchains of a linear relation ordering, instead of arbitrary combinations of relations. An example for that is shown in Figure 4. It shows a query graph together with a linearized representation of the same graph. For the original graph a DP algorithm would fill a table with 17 entries, while if we restrict the DP algorithm to consider the relations only in linearized order, the DP algorithm fills a table with only 6 entries. Note that this difference grows exponentially with size of the query, in general it is $O(2^n)$ for the full DP and $O(n^2)$ for the linearized DP. Unfortunately, hyper-graphs cannot be expressed in such a linearized form. Thus, linearized DP can only be applied to queries that can be represented by a regular graph.

While this linearization greatly reduces optimization time, the way we linearize the query graph clearly has a large impact on the quality of the final plan. If we pick a bad order here, some good join orders become impossible during the dynamic programming phase,

```

linearizedDP( $Q = (V, E)$ )
// constructs a bushy join tree for the query  $Q$ 
// find a linearization using IKKBZ
 $Q' = \text{minimal spanning tree of } Q$ 
 $O = \text{IKKBZ}(Q')$ 
label nodes  $v$  in  $V$  from 0 to  $|V| - 1$  as ordered in  $O$ 
// find the optimal plan for the linearization
 $T = \text{empty DP table of size } |V| * |V|$ 
for  $v_i \in V$ 
   $T[i, i] = v_i$ 
for  $s = 2 \dots |V|$ 
  for  $i = 0 \dots |V| - s$ 
    for  $j = 1 \dots s - 1$ 
       $L = T[i, i + j - 1], R = T[i + s, i + s - 1]$ 
      if  $L$  can join with  $R$ 
         $P = L \bowtie R$ 
        if  $C(P) < C(T[i, i + s - 1])$ 
           $T[i, i + s - 1] = P$ 
return  $T[0, |V| - 1]$ 

```

Figure 6: Linearization in Combination with DP

and we might lose optimality. Fortunately, we know a very good way to order relations: The IKKBZ algorithm [13, 15] can find the optimal left-deep tree for an acyclic query graph in polynomial time (see Figure 5). It tries to sort all joins by *rank*, i.e., the cost/benefit ratio. For C_{out} that is $\frac{1 - sel}{costs}$. If sorting by rank is not possible due to the restrictions of the query graph, the *IKKBZ-normalize* step collapses such contradictory sequences into compound relations until all relations are sorted by rank. It has been shown that this algorithm produces the optimal left-deep tree for acyclic query graphs with ASI cost function. And a left-deep tree is a linearization, of course. Thus, if a query has up to 100 relations, but is too complex for the *small* case from the previous section, we linearize its query graph by running IKKBZ. Note that while IKKBZ gives us the optimal left-deep tree, we consider bushy trees here, and thus IKKBZ does, in general, not produce the optimal join tree for our setting. Nevertheless, the linearization we get from it is very good and guarantees desirable properties for the final join tree: If the optimal join tree is indeed linear (left/right-deep or zig-zag), the solution generated by the linearized DP is guaranteed to be optimal. If the optimal tree would be bushy instead, the cost of the optimal linear tree is an upper bound for the costs of the final plan. Note that this ensures optimal solutions for the entire class of star queries as their solutions are inherently linear.

If the query graph is cyclic we construct a minimum spanning tree (minimizing the join selectivities) before running IKKBZ (as described in [15]). Note that this spanning tree is only used for IKKBZ itself, because IKKBZ needs an acyclic query graph. The subsequent DP step uses the full query graph, including cycles. As we will see in Section 6, the linearization with IKKBZ leads to excellent results, often preserving optimality or being near optimal.

The full algorithm is shown in Figure 6. It first runs IKKBZ to linearize the relation order, and then runs a dynamic programming algorithm to construct the optimal bushy plan for that linearization. Note that the DP algorithm is different from normal DP algorithms

```

GOO-DP( $Q = (V, E), dp, k, budget$ )
// use GOO to guide the  $dp$  algorithm to problems of size  $k$ 
// construct a bushy plan using GOO
 $T = V$ 
while  $|T| > 1$ 
   $(L, R) = \text{argmin}_{L, R \in T, L \text{ can join } R} |L \bowtie R|$ 
   $T = T \setminus \{L, R\} \cup \{L \bowtie R\}$ 
   $T = \text{pick the single element of } T$ 
// run  $dp$  on problems up to size  $k$ 
while budget > 0
  pick  $T' \in T$  such that size of  $T' \leq k \wedge$ 
    size of  $T'.parent > k \wedge C(T')$  is maximal
  replace  $T'$  with  $dp(T')$ . Consider new  $T'$  as base relation.
  reduce budget by DP table size from last  $dp$  call
return  $T$ 

```

Figure 7: Using GOO to guide a DP algorithm

like DPSize, it is a custom algorithm for the linearized query graph. The idea of the DP phase is that the relations form a linear chain, and thus every subproblem that we solve must be a subchain of that linearization. We can thus identify each problem with the start and end node of that chain, and accordingly the DP table T is organized by $|V| \times |V|$. After filling that table we can extract the optimal solution for that linearization from $T[0, |V| - 1]$. Note that this DP strategy explicitly exploits the fact that our search space is linear, and that it is much simpler (and thus faster) than general DP strategies that have to handle arbitrary query graphs.

The linearization step makes DP tractable for much larger queries, but still there are limits. The DP phase has a runtime of $O(n^3)$, which is fine for up to 100 relations (or more on a fast machine), but at some point we must switch to an even cheaper algorithm. We discuss the optimization of such large queries next.

4.3 Large Queries

When the query becomes too large for the $O(n^3)$ algorithm from the previous section, we have to introduce a greedy step. Here, we use an idea from Iterative DP [14]: We first construct an execution plan using a greedy algorithm, and then improve that plan by running a more expensive optimization algorithm on subplans up to size k . The nice thing about that approach is that we can control the optimization time by changing k , interpolating between greedy and full optimization. A limitation however is that the more complex optimization cannot go outside the current subtree, i.e., we cannot move a relation more than k positions within the tree.

This was problematic for the original Iterative DP paper, as they had to choose relatively small values for k (e.g. 4–7) to keep optimization time reasonable. We however use the linearized DP from the previous section as expensive optimization step, which allows us to choose $k = 100$. This way the DP phase has much more freedom to correct mistakes the greedy phase has introduced, as now relations can move up to 100 places within the tree. For the greedy phase we use Greedy Operator Ordering (GOO) [8], as GOO produces good bushy plans, and can be implemented efficiently.

The resulting algorithm is shown in Figure 7. It first runs GOO to construct a bushy plan, and then runs a dynamic programming

```

adaptive( $Q = (V, E)$ )
// optimize the query  $Q$  adaptively, depending on its complexity
// find the optimal solution if possible
if  $|V| < 14 \vee \text{countCC}(Q, 10000) \leq 10000$ 
    return DPHyp( $Q$ )
// use linearized DP to handle large queries
if  $Q$  contains no hyper-edges
    if  $V \leq 100$ 
        return linearizedDP( $Q$ )
    return GOO-DP( $Q$ , linearizedDP, 100, 10000)
// use DPHyp as inner DP algorithm for hyper-graphs
return GOO-DP( $Q$ , DPHyp, 10, 10000)

```

Figure 8: Adaptive Optimization Algorithm that Considers the Query Complexity

algorithm on the most expensive subtree of up to size k until the optimization budget is exhausted. After each iteration we reduce the optimization budget by the size of the DP table of the inner DP algorithm, which allows us to control the overall optimization time.

Usually we run GOO-DP as GOO-DP(Q , linearizedDP, 100, 10000), which gives very good results with modest optimization time, as we will see in Section 6. Only if the query graph contains hyper-edges due to non-inner joins we cannot easily utilize linearizedDP, as our linearization algorithm assumes a regular graph. If the query graph contains hyper-edges we instead call GOO-DP as GOO-DP(Q , DPHyp, 10, 10000), which uses the hyper-graph aware DPHyp algorithm as inner DP algorithm. While correct, the downside is that we have to choose a much smaller k value then. In future work we therefore plan to generalize the linearization to hyper-graphs, even though it affects only relatively few queries.

4.4 Putting Everything Together

When given a query Q , our *adaptive* optimization strategy picks the most appropriate algorithm depending on the complexity of the query (shown in Figure 8). If the query graph contains up to 10,000 connected subgraphs we know that graph-based DP strategies are fast, so we can use DPHyp to optimize the query. If the number of relations is less than 14 this is always the case, regardless of the query structure and we can skip counting then. Thus we get the optimal join order for such simple queries by running DPHyp.

Otherwise, we first check if the query contains hyper-edges due to non-inner joins. Usually that is not the case, and we optimize it using linearizedDP if the query contains up to 100 relations. For even larger queries we use GOO to direct the linearizedDP optimization to problems up to size 100. For queries with hyper-edges we do the same, but we have to use DPHyp as inner optimization algorithm and thus consider only problems up to size 10.

A nice property of that adaptive strategy is that it is guaranteed to find the optimal solution for the common case of reasonably simple queries, but at the same time scales smoothly up to mega queries with thousands of relations. We avoid a hard break in plan quality along that way by 1) switching to the less expensive but still very powerful linearized DP when the query grows, and then 2) by optimizing very large subproblems of up to 100 relations with DP if we have to introduce a greedy step.

5 IMPLEMENTATIONS FOR LARGE QUERIES

In the previous section we have introduced our adaptive join ordering framework which efficiently handles a wide range of queries. Once queries become reasonably large, not only the join ordering algorithm itself plays an important role, but the implementation of datastructures and the corresponding algorithms have noticeable impact on optimization times. In this section we thus take a closer look onto some implementation details necessary to ensure maximum performance.

5.1 Representing Sets of Relations

An important datastructure for all classical join ordering algorithms are sets of relations. It is thus crucial to handle such sets as efficient as possible. We refer to such sets as *BitSets* and use four different implementations: for up to 64 relations a 64 bit unsigned integer is used, where the k -th bit represents the presence of relation k in the set. Integers allow to perform set operations such as union with a single machine instruction. For up to 128 relations, we use two integers requiring two instructions for union or intersection. For larger queries, a vector of integers is used, where the presence of relation k in the set is encoded by the bit at position $k \bmod 64$ being set in the $\lfloor \frac{k}{64} \rfloor$ -th integer. Union and intersection are still bitwise operations here, but we have to perform them for all integers in the vector. A *sparse BitSet*, that maintains a sorted vector of relations, is employed for queries with more than 1024 relations. For such a set, union and intersection become much more expensive to compute.

5.2 Representing Join Edges

Cardinality estimates are the most important measure to determine the cost of a certain query plan. Initially, only the cardinalities of the involved base relations and the selectivities of join edges are known. During join order optimization, the result cardinalities of joins have to be estimated. As long as a query is acyclic, independence between predicates is assumed and the cardinality of an intermediate result is estimated as the product of the cardinalities of the covered relations and the respective selectivities of the involved join predicates. Although this independence assumption has been shown to systematically underestimate the cardinalities of intermediate results, it is widely used by existing database systems [17]. Providing more accurate estimates for intermediate results of joins is still a major research topic [18].

Cardinality estimates, however, become unacceptably inaccurate for cyclic queries. To dampen this effect, we build the minimum spanning tree (MST) of the involved joins during estimation with regards to their selectivity and estimate the result cardinality based on this MST. We employ Kruskal's algorithm [16] and utilize a union-find datastructure to efficiently build the MST. As this is expensive, we only build the MST at the moment we actually build a cyclic partial solution. Efficient cycle detection is achieved by 1) utilizing a union-find datastructure to recognize redundant join edges, and 2) by introducing a *join lookup* table that gives us all joins touching a particular relation. By using that lookup for the relations of the smaller side of a join, we can quickly find all relevant joins.

Both, union-find and the join lookup are not only necessary for efficient cycle detection and cardinality estimation, but can also be utilized to directly speed up all cases where we have to find connecting join edges (e.g. GOO [8] and QuickPick [31]).

6 EVALUATION

We have evaluated our approach over a large spectrum of queries, and compared it with many competing algorithms, including commercial database systems. In the following, we first discuss the experimental setup, and then show results for standard benchmarks like TPC-H and for scalability experiments using very large synthetic workloads. Finally we show the effect of individual implementation techniques on the performance of various algorithms.

6.1 Experimental Setup

For the following experiments we extracted the query graphs of the individual workloads, including cardinalities and selectivities, using the encoding schema A from [20] to represent non-inner join edges. We then passed these query graphs to the various algorithms to construct join trees. Each algorithm was run with a timeout of 60s, i.e., if we report an optimization time of 60s the algorithm was unable to find a solution in the given time frame. As cost function we used C_{out} , and we report all costs normalized to the *best non-cross-product solution* found. As long as at least one of the DP algorithms terminates this is guaranteed to be the optimal solution, i.e., then normalized costs of 1 indeed means optimality. For large queries the optimal solution is simply unknown, therefore then normalized costs of 1 only means that the plan is the best plan found, but that is still useful to compare algorithms. Note that MILP might produce a solution with normalized cost less than 1 if it deems cross-products profitable. The experiments were run on a 4 socket Xeon E7-4870 v2 / 2.30 GHz machine, running Ubuntu 17.04. The algorithms were compiled with gcc 6.3, using O3 and march=corei7.

As competitors, we implemented a wide range of algorithms, starting from the classical System R-style dynamic programming over the size of the join tree [24], implementing both a left-deep variant (*DPSizeLinear*) and a bushy-tree algorithm (*DPSize*). As an example of a graph-based DP algorithm we implemented *DPHyp* [22], which has better asymptotic behavior than *DPSize*. To cope with large queries we added the *simplification* algorithm [23] to make them tractable for *DPHyp*. We also implemented the mixed integer linear programming approach [30] and used Gurobi [10] to solve the MILP and obtain a linear join tree. As greedy heuristic we started with the *minsel* heuristic [27] that orders joins increasing in selectivity. Greedy Operator Ordering [8] (*GOO*) is a more advanced greedy algorithm that can construct bushy plans and often works quite well. We combine *GOO* and *DPHyp* using Iterative DP (more precisely: IDP-2) [14] and call the resulting algorithm *GOO/DP*. IDP-1 would not be suited for the query sizes that we consider. Furthermore, we implemented the IKKBZ algorithm [13, 15], which can find the optimal left-deep tree for acyclic query graphs in polynomial time. If the query graph is cyclic, we construct a minimum spanning tree over the edge selectivities first. If the query graph contains hyper-edges due to non-inner joins, we fall back to *GOO* instead, as the IKKBZ algorithm cannot handle hyper-edges (this happens only for a handful of queries in our workloads). As a randomized approach we implemented *QuickPick* [31] to construct 1,000 random join trees, and then pick the cheapest one. Finally, we added a genetic algorithm (*genetic*) for join ordering, similar to the genetic algorithm strategy of PostgreSQL.

Table 1: Total Optimization Time and Geometric Mean of Normalized Costs in Some Popular Benchmarks

total optimization time (ms) / geomean of normalized costs					
benchmark	TPC-H	TPC-DS	LDBC	JOB	SQLite
minsel	<1 / 1.06	<1 / 1.03	<1 / 1.04	3 / 1.03	958 / 1.00
GOO	<1 / 1.05	<1 / 1.01	<1 / 1.02	<1 / 1.05	<1 / 1.00
DPSize	<1 / 1.00	479 / 1.00	<1 / 1.00	417 / 1.00	33K / 1.00
DPSizeLin.	<1 / 1.04	36 / 1.03	<1 / 1.04	100 / 1.02	1.8K / 1.00
MILP	698 / 1.05	7.7K / 0.92	845 / 0.98	290K / 1.58	5.1M / 1.00
DPHyp	<1 / 1.00	128 / 1.00	<1 / 1.00	213 / 1.00	2.1K / 1.00
IKKBZ	<1 / 1.02	<1 / 1.07	<1 / 1.02	<1 / 1.17	520 / 1.00
linDP	<1 / 1.00	<1 / 1.00	<1 / 1.00	3 / 1.07	4.7K / 1.00
GOO/DP	<1 / 1.00	1 / 1.00	<1 / 1.00	1 / 1.00	23 / 1.00
GOO/linDP	<1 / 1.00	1 / 1.00	<1 / 1.00	3 / 1.05	5.0K / 1.00
QuickPick	<1 / 1.00	82 / 1.00	3 / 1.00	113 / 1.01	3.8K / 3.10
Genetic	556 / 1.00	5.6K / 1.00	751 / 1.00	6.6K / 1.02	56K / 4.10
Simpl.	<1 / 1.00	17 / 1.02	<1 / 1.00	112 / 1.00	2.2K / 1.00
adaptive	<1 / 1.00	5 / 1.00	<1 / 1.00	53 / 1.00	2.3K / 1.00

For our own approaches, we implemented the search space linearization (*linearizedDP*), the iterative DP combination of *GOO* and *linearizedDP* (*GOO/linDP*), and our adaptive strategy (*adaptive*) that switches between algorithms based upon query complexity.

All algorithms were carefully implemented using the implementation tricks from Section 5. In particular, we tried to make sure that all observed runtime differences stem purely from differences in the asymptotic behavior, and not from quality-of-implementation issues. We also compared with existing implementations if possible, and for example our *GOO* implementation is about twice as fast as the already quite cleverly implemented original *GOO* version from <http://lambda.uta.edu/order/>. All algorithms were extended to handle hyper-edges except IKKBZ, which runs *GOO* in that case.

6.2 Standard Benchmarks

We tested our algorithms with a wide variety of use cases, and started with well known standard benchmarks. Unfortunately, most benchmark queries are too small for the purpose of this paper and can easily be solved exactly using *DPHyp*. Nevertheless, we briefly discuss them here for completeness.

We started with the TPC-H benchmark [29]. From its 22 queries we extracted 23 query graphs (some queries contain multiple join blocks and some queries contain no joins at all), with a maximum graph size of 8 relations and a median size of 3 relations. Accordingly, the join ordering problem is very easy to solve, the results are shown in the first column of Table 1. We report the total optimization time for all 23 query graphs, followed by the geometric mean of the normalized costs. All algorithms finish in less than 1ms (except *genetic* and *MILP*), and most of them find optimal solutions. Of the 343 join plans constructed by all the algorithms only 22 were suboptimal, with maximum normalized costs of 2.98 for one *GOO* plan. Clearly, TPC-H is no challenge for join ordering algorithms.

The TPC-DS benchmark [28] contains much more, and much more complex queries. From the 99 TPC-DS queries we extracted 236 query graphs, with a maximum of 18 and a median of 3 relations. Thus, even though TPC-DS contains some large queries, most of

them are still small. Only six query graphs have more than 8 relations. Nevertheless, they become visible in column two of Table 1. Most algorithms are still very fast, but the DP based algorithms, genetic and MILP increase optimization time, mainly due the few large queries. Note that the reported time is the total time for all 236 query graphs, thus optimization time is still negligible here. Also note that normalized costs of the MILP solvers solutions are less than 1 here, as cross products are beneficial for a few expensive queries (none of the other algorithms considers cross products).

Another benchmark we considered is LDBC BI [1] that evaluates analytical queries on social network data. We extracted 55 query graphs, with a maximum size of 13 relations and a median size of 3 relations. Only 3 query graphs have more than 7 relations, and accordingly, all algorithms handle the queries without problems.

The Join Order Benchmark (JOB) [17] aims to stress the query optimizer, but unfortunately it mainly stresses the quality of cardinality estimation, not the optimization algorithm itself. From the 113 queries we got 113 query graphs, with a maximum size of 17 relations and a median size of 8 relations. This is significantly larger than the other benchmarks, but due to the exponential nature of the optimization problem it is overall not much more expensive than the TPC-DS benchmark with its 18 relation join query.

Another interesting benchmark is included in the SQLite test suite [12]. The `select5`.test file contains join queries of increasing size, with up to 64 relations. We extracted 732 query graphs, with a maximum of 64 relations and a median of 34 relations. This is the only publicly available workload that we are aware of with such large queries, and it is thus very interesting for join ordering. But unfortunately all joins are PK/FK joins and all queries contain a filter predicate that ensures that one can evaluate all queries with intermediate result sizes of 1 when following the PK/FK structure. Accordingly, all algorithms always find the optimal plan, except the randomized strategies *QuickPick* and *genetic*. Still, the optimization time itself is interesting (see fourth column of Table 1). Using the MILP solver took more than 84 minutes to optimize the 732 queries. As the MILP considers cross products, it does not benefit from the simplicity of the query graphs. The dynamic programming strategies start to have problems, with over 33s for *DPSize*. *DPHyp* handles the workload gracefully because the query graph structure is very benign (a chain) and *DPHyp* has polynomial performance for these cases. It even outperforms *linearizedDP* here because *DPHyp* avoids enumerating disconnected subgraphs and handles commutativity, while *linearizedDP* considers all $O(n^2)$ possible subchains, independent of connectedness. Note however that this is only a constant factor difference. The adaptive strategy (correctly) picks *DPHyp* for all test cases, thus guaranteeing optimality, and only adds a few percent overhead to check the query complexity.

6.3 Scalability Experiments

The queries in the standard benchmarks are too small to highlight the asymptotic behavior of the different algorithms. We have access to large queries from commercial workloads, but unfortunately these are not publicly available. And furthermore, there are too few of them, at least on the upper end of the spectrum. We are only aware of a handful of queries with more than 1000 relations, but that number is too small for meaningful experiments. Instead, we

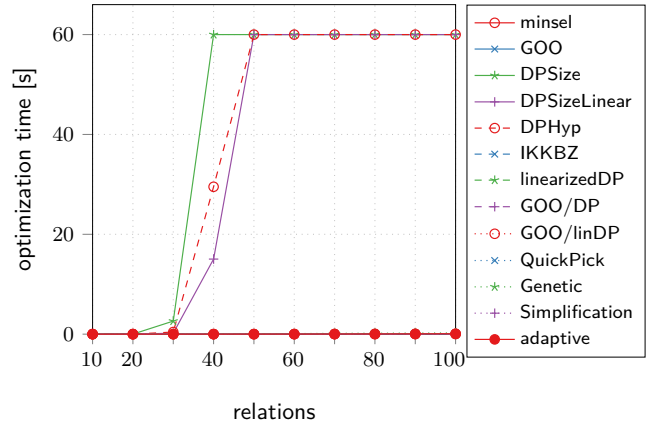


Figure 9: Median Optimization Time for Random Tree Queries of Size 10-100 (100 queries per size)

generate synthetic queries for scalability experiments. We follow the procedure from [23] to generate realistic tree queries with 10 to 5,000 relations (most edges are PK/FK joins, a few are FK/FK joins, relation sizes vary, etc.). For every size we generate 100 different queries and optimize all of them using the various algorithms. Note that tree queries are known to be NP-hard in general [3], and accordingly the DP algorithms exhibit exponential runtime. In order to get meaningful plots, we look at different size ranges in sequence.

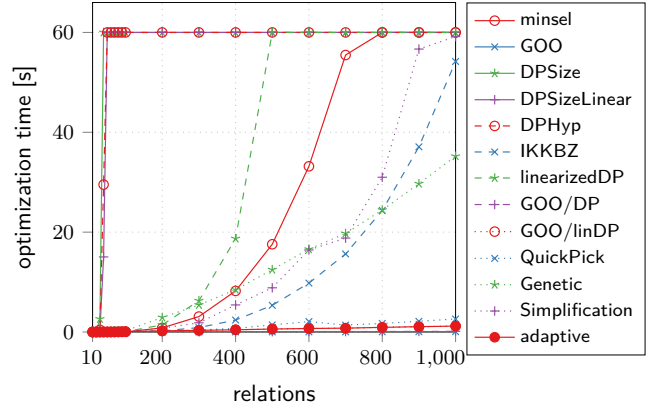
We start with small and medium sized queries with sizes up to 100 relations. These make up more than 99% of all queries in most workloads, and this size class is therefore by far the most important one. The median optimization time of the different algorithms is shown in Figure 9. All algorithms time out after 60s, thus optimization times are capped at 60s. Note that we only report median optimization times here to keep the plots readable. For a more detailed look at the distribution we refer to Table 5–9 in the appendix. In this plot we mainly observe two things: 1) the DP based algorithms fail at some point due to the exponential nature of the problem. The exact point depends on the algorithm, *DPSize* starts to fail for queries with 30 relations, while *DPHyp* can still optimize 69% of the queries with 40 relations within one minute, but for larger query sizes DP is not an option. *DPHyp* can solve only 2% of the queries with 50 relations and none of the larger queries. *DPSizeLinear* is about the same (85% of size 40, 5% of size 50), and of course it only considers a restricted search space. For query sizes less than 40, however, DP is a perfectly viable strategy, as it guarantees finding the optimal solution, and optimization time is not too high, at least for *DPHyp*. The MILP solver performs even worse than the DP algorithms as it considers cross products. It starts experiencing timeouts for queries with 20 relations and starting with some queries of size 40 it is unable to provide any solution at all. Thus we can and should use dynamic programming for small queries, but we have to switch to something else for larger queries. 2) The other alternatives all do fine with up to 100 relations. Optimization times for 100 relations are typically between 10ms and 70ms, and thus insignificant compared to the query size.

However, optimization time is only one aspect when comparing algorithms, the other question is how good the constructed plans

Table 2: Relative Costs for Random Tree Queries of Sizes 10-100 (100 queries per size)

number of relations	normalized costs (avg / 95% / max)					
	10	20	30	40	70	100
minsel	5.0 / 6.3 / 309.2	44.5 / 35.1 / 2.8e3	1.0e3 / 292.4 / 8.7e4	125.5 / 160.6 / 1.0e4	308.4 / 1.7e3 / 9.6e3	2.0e4 / 1.2e5 / 1.0e6
GOO	1.0 / 1.2 / 1.6	1.1 / 1.7 / 2.9	1.3 / 2.4 / 6.8	1.2 / 1.9 / 2.5	1.4 / 2.5 / 6.1	1.4 / 2.6 / 4.5
DPSize	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0		
DPSizeLinear	1.6 / 3.0 / 23.7	2.1 / 4.1 / 58.4	1.5 / 2.0 / 27.9	1.2 / 2.0 / 4.9		
MILP (optimal)	2.3 / 8.1 / 23.7	2.8 / 7.1 / 58.8	2.0 / 4.6 / 16.8	1.6 / 3.5 / 3.5	1.1 / 1.3 / 1.3	
DPHyp	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0		
IKKBZ	1.6 / 3.0 / 23.7	2.1 / 4.1 / 58.4	1.5 / 2.0 / 27.9	1.2 / 2.0 / 4.9	1.0 / 1.3 / 1.6	1.0 / 1.2 / 1.5
linearizedDP	1.0 / 1.0 / 1.3	1.0 / 1.4 / 1.8	1.0 / 1.3 / 2.2	1.0 / 1.2 / 1.5	1.0 / 1.0 / 1.3	1.0 / 1.0 / 1.0
GOO/DP	1.0 / 1.0 / 1.0	1.1 / 1.5 / 2.9	1.2 / 2.4 / 6.7	1.1 / 1.7 / 2.5	1.2 / 2.2 / 4.0	1.3 / 2.1 / 4.5
GOO/linDP	1.0 / 1.0 / 1.0	1.0 / 1.4 / 1.8	1.0 / 1.3 / 2.2	1.0 / 1.2 / 1.5	1.0 / 1.0 / 1.3	1.0 / 1.0 / 1.0
QuickPick	1.0 / 1.0 / 1.1	1.2 / 2.2 / 3.3	5.5 / 22.6 / 54.7	9.9 / 35.5 / 120.8	162.1 / 291.2 / 9.4e3	248.4 / 887.8 / 3.8e3
QuickPick (60s)	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.10	1.3 / 2.0 / 19.5	2.1 / 6.1 / 12.5
Genetic	1.0 / 1.0 / 1.0	1.0 / 1.2 / 5.0	1.2 / 2.2 / 3.8	1.3 / 2.5 / 10.5	2.8 / 11.5 / 17.9	5.6 / 18.6 / 42.0
Genetic (60s)	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.5	1.1 / 1.4 / 3.9	1.1 / 1.2 / 3.4	1.3 / 2.6 / 6.5	1.4 / 3.0 / 17.8
Simplification	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.9	1.1 / 1.9 / 2.9	104.2 / 6.6 / 1.0e4	3.5e6 / 342.7 / 3.4e8	1.3e4 / 9.3e3 / 6.6e5
adaptive	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.4	1.0 / 1.3 / 2.2	1.0 / 1.2 / 1.5	1.0 / 1.0 / 1.3	1.0 / 1.0 / 1.0

are. In Table 2 we show for all algorithms the average costs (normalized to the best plan found), the 95% quantile and the maximum costs. For the MILP solver we only report normalized costs, if it did not time out, as quality degrades significantly once timeouts happen. The DP strategies obviously always have normalized costs of 1, they always find the optimal solution. DPSizeLinear is mostly ok, but sometimes it has higher costs, up to factor 58 worse than DPHyp, because it only considers left-deep plans, which are inferior to bushy plans. The same is true for IKKBZ. The quality of the solutions obtained from the MILP solver are comparable with the ones constructed by DPSizeLinear. However, as soon as the MILP solver stops early due to timeout, the plan quality gets significantly worse. We routinely observe plans orders of magnitude worse than the best known plan in case of timeouts (not shown in the Table). From the heuristics minsel performs very poorly, often constructing very bad plans with normalized costs of up to 10^6 . The GOO algorithm is much more reasonable, constructing good plans even for large queries. It is not ideal, though, with a maximum normalized costs of 6.8. The iterative GOO/DP variant performs basically the same, the plans are slightly better but the differences are a few percent. Query simplification works well as long as the query is reasonably small, but for large queries it is forced to heavily rely upon its greedy step, resulting in poor plan quality. QuickPick shows similar behavior, it works well for small queries, but as the query size grows, quality degrades. The genetic algorithms performs ok, but costs are also quite high in some cases, up to 42 for 100 relations. These randomized algorithms can of course simply be run for a longer period of time to improve their result. Running Quickpick and the genetic algorithm for 60 seconds (which is much more than our adaptive framework requires) resulted in better plans (Quickpick (60s) and Genetic(60s) in Table 2). The number of plans considered however, increases linearly with the runtime, whereas the search space grows exponentially with the size of the query. Thus, regardless of how long the randomized algorithms run, starting at some size, they will simply explore too few plans to discover one of the few good plans.

**Figure 10: Median Optimization Time for Random Tree Queries of Sizes 10-1000 (100 queries per size)**

Our own linearized DP algorithm works well over the whole spectrum, the mean is 1.0 (i.e., perfect), the 95% costs are at most 1.3, and the maximum costs are 2.2. And that with a tiny fraction of the optimization time of a full blown DP algorithm. Note that for this size class GOO/linDP is identical to linearizedDP, the linDP step optimized the whole tree. The adaptive strategy switches between DPHyp and linearizedDP here before the optimization time of DPHyp would become noticeable. When interpreting the cost numbers, note that we normalize to the best plan found. Up to size 40 that includes the optimal solution in nearly all cases, i.e., the costs are correct, but for larger queries we can no longer guarantee optimality due to the NP-hardness. This is the reason why the linearizedDP algorithm (and correspondingly GOO/linDP and adaptive) seems to get better as the problem size increases: At size 100, it is simply the best algorithm around as the DP algorithms fail. But as we have seen for the smaller sizes, it works very well in general, even in cases where regular DP would have been an option.

Next, we look at queries with 100 to 1,000 relations. These are rare, but they occur from time to time, and they tend to be expensive. Thus, we need to be able to handle them reasonably well, concerning both the quality of the generated plan and optimization time. We show the optimization times in Figure 10. The regular dynamic programming strategies already failed for smaller queries, but for this size class most of the other algorithms tend to have problems. The linearizedDP algorithm has $O(n^3)$ runtime, and thus starts to get expensive for more than 200 relations. Minsel, simplification and IKKBZ also start to timeout at different sizes. The genetic algorithm can still optimize 1,000 relations, but it already needs 35s. Only QuickPick and the various GOO variants can handle these sizes reasonably (including our *adaptive* strategy, which switches to GOO/linDP for this size). Concerning result quality, QuickPick deteriorates (median of 10^6 , maximum 10^{14}). GOO is typically doing ok (median 1.48), but generates poor plans, too, from time to time (e.g., maximum cost of 19.1 for one query of size 800). The iterative DP variant GOO/DP is basically identical. Our combination of GOO with linearizedDP (GOO/linDP) is doing better, because it combines GOO with a much larger DP correction. Its median costs are 1.00 (maximum cost of 3.89 for size 800). The maximum cost is not perfect, but still ok, and much better than all alternatives.

The plot also shows that just abandoning optimality is not enough to handle large queries. Even when implementing heuristics, it can easily happen that optimization does not terminate in reasonable time when queries become large. We will also see that below when looking at existing systems. This emphasizes that one needs to carefully implement scalable algorithms, potentially switching between alternatives multiple times, based upon the query complexity.

Finally, we look at the largest size class, containing queries with 1,000 to 5,000 relations. These are unique beasts that occur very rarely, but they occur in practice [19] (see Figure 12 in the appendix for a plot of the optimization times). Only QuickPick and the GOO variants can handle this size category (and *adaptive*, that always uses GOO/linDP here). QuickPick has very poor plan quality (median costs of 10^{10}), GOO is much better. GOO/linDP has slightly higher optimization costs due to the extra linearizedDP step, but the generated plans are better, too, with lower average and maximum costs. The median normalized costs of GOO/linDP are 1.0, but that just means that it is the best algorithm around. For these sizes of queries we must be happy if we can construct reasonable plans at all, and as mentioned in [7] this is not the case for most optimizers.

Overall our adaptive strategy handles all size classes very well, choosing the appropriate algorithm based upon the query complexity, and consistently combining the best generated plans with low optimization time over the whole spectrum of query sizes.

6.4 Cost Models

So far, we evaluated our framework using the rather simplistic C_{out} cost function. In this section we investigate, whether the results presented so far also hold true if we use more advanced cost models.

We first investigate the well known cost model for grace-hash join from [11]. This cost function models IO in great detail. The following table shows the distribution of the relative costs of the resulting plans (see Table 10 in the appendix for more details).

Query Set	1	(1,1.1]	(1.1,2]	>2
standard benchmarks	100%	0%	0%	0%
generated	73.2%	26.3%	0.2%	0.3%

Our framework gives the optimal solution for all the standard benchmark queries. For the generated tree queries, 73.2% of the plans are optimal, 26.3% are within 10% of the optimum, four of the 2,300 plans are worse by up to a factor of 2 and seven plans are worse than that.

Another interesting cost model is C_{mm} [17] which accounts for indexes. This model is particularly interesting, as it cannot be used for ranking in the IKKBZ algorithm. We thus use C_{out} for ranking but use C_{mm} for the other algorithms and the DP phase of linearized DP. For this experiment, we randomly enabled indexes for 25% of the joins. As can be seen from the following table even without any additional effort to make the linearization aware of indexes, plan quality is still very good (see Table 11 in the appendix for more details).

Query Set	1	(1,1.1]	(1.1,2]	>2
standard benchmarks	99.9%	0.1%	0%	0%
generated	65.1%	34.5%	0.4%	0%

All plans for the standard benchmarks are optimal except one which is suboptimal by at most 10%. The generated queries get decent plans. 65.1% of them are optimal, 34.5% worse by at most 10% and 0.4% at most twice as expensive as the best known plan.

The results of experiments with different index configurations (100%, 50%, 10% and 1%) are similar to those reported here.

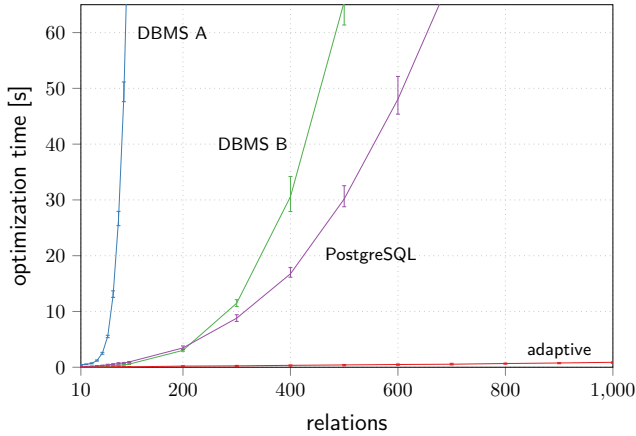
6.5 Cardinality Estimates

Cost based optimizers largely rely on cardinality estimates to determine the cost of a query plan. Usually, cardinalities of base relations can be estimated quite good. However, even small errors in these estimates can cause enormous errors to estimates of intermediate results of joins [17]. In the following, we thus investigate the effect of errors in cardinality estimates on the quality of the plans generated by our adaptive join ordering framework. We introduce gamma distributed random noise to the cardinalities of base tables and join selectivities and run the different optimization algorithms using these distorted cardinalities. For the resulting plans, we calculate the true costs using the true cardinalities and report the normalized true costs for the most prominent algorithms in Table 3.

As expected, the larger queries become, the more the algorithms suffer from the bad estimates. Minsel again results in many plans suboptimal by several orders of magnitude. While QuickPick gives relatively decent plans as long as queries are small, for larger queries its quality drops significantly. Even dynamic programming, while mostly doing good, sometimes results in plans being suboptimal by a factor of up to 14.8. The quality of our adaptive framework is also influenced by the bad cardinality estimates. Nevertheless, its plans are decent throughout the whole spectrum of query sizes compared to the other heuristics. These results suggest, that applying advanced join ordering algorithms is still beneficial despite bad cardinality estimates, as the cheap heuristics commonly applied to such large queries, in general, result in clearly inferior plans.

Table 3: Relative Costs with random noise on base table cardinality estimates and join selectivities for Random Tree Queries of Sizes 10-100 (100 queries per size)

number of relations	normalized true costs (avg / 95% / max)				
	10	20	30	40	100
minsel	7.6 / 6.9 / 309.2	44.9 / 51.6 / 3.6e3	1.5e3 / 2.0e3 / 8.7e4	188.3 / 1.2e3 / 1.0e4	501.1 / 4.6e3 / 9.7e3
GOO	1.1 / 1.5 / 2.0	1.4 / 3.3 / 11.3	1.6 / 3.1 / 7.4	3.0 / 16.7 / 54.0	6.4 / 34.2 / 105.0
DPHyp	1.1 / 1.4 / 2.0	1.1 / 1.7 / 2.3	1.3 / 3.3 / 4.0	1.4 / 1.9 / 14.8	
QuickPick	1.1 / 1.4 / 2.0	1.5 / 3.1 / 4.4	8.4 / 35.1 / 101.0	15.9 / 90.0 / 142.0	355.9 / 1.4e3 / 9.5e3
adaptive	1.1 / 1.4 / 2.0	1.1 / 2.0 / 2.3	1.3 / 2.3 / 4.0	1.7 / 3.2 / 17.5	4.2 / 18.0 / 97.4

**Figure 11: Comparison with Existing Systems; Random Tree Queries of Size 10-1000 (100 queries per size)**

6.6 Other Systems

As most of the investigated algorithms already fail to optimize queries of modest size in reasonable time, we now investigate how existing database systems cope with larger queries and if they implement fast algorithms to optimize them. We compare our adaptive join ordering algorithm with two commercial database systems and the open source system PostgreSQL. Note that we only report compile times here, as the optimization goals vary between the systems and thus there is no meaningful way to compare the quality of the plans generated by the different systems.

If available for a system we ran tools similar to PostgreSQLs explain to obtain compilation times. Some of the systems, however, are not reporting such timings even if they offer plan explanation. In these cases we measured the response time of the explain tool.

The median optimization times of these systems for the set of generated queries with up to 1,000 relations are shown in Figure 11. The error bars indicate the range from minimum to maximum optimization time. All of the investigated systems start struggling once the queries contain a few hundred joins. For DBMS A, compile times are already increasing noticeably on queries with 50 relations. This system fails to compile queries containing 100 relations within one minute. DBMS B is able to compile queries with 300 relations in about 10 seconds. For queries larger than 500 relations, it is no longer able to provide a plan within one minute. Similarly, the fastest of the investigated systems, PostgreSQL, which uses genetic

algorithms for large queries, is able to optimize queries with 300 relations in less than 10 seconds. However, its compilation times exceed one minute for queries with 700 relations, a size where our adaptive approach takes about 500 milliseconds. From these results we conclude, that neither the algorithms nor the data structures used by the investigated systems are tailored towards efficient handling of such large queries.

6.7 Linearized DP

Having shown the quality and performance of our adaptive framework over a wide spectrum of queries, we now take a detailed look at the properties of the novel linearizedDP presented in Section 4.

Quality. Looking at the quality of the plans generated by linearizedDP we distinguish three sets of queries: those extracted from the standard benchmarks (for all of them the optimal solution is known), the generated queries where an exact algorithm finished and thus the optimal solution is known and the generated queries where linearizedDP finished successfully but an optimal plan is not known. In the following table we report the distribution of the normalized costs of the plans constructed by linearizedDP:

Query Set	1	(1,1.1]	(1.1,2]	>2
standard benchmarks	1,127	16	13	3
gen. (opt. known)	238	95	37	1
gen. (opt. unknown)	919	31	20	1

LinearizedDP finds the optimal plan for 1,127 (97%) of the queries extracted from the standard benchmarks. For 16 (1.3%) queries, the costs of the plan constructed by linearizedDP is within 10% of the optimal plan. Another 13 (1.1%) plans are suboptimal up to a factor of two and only 3 plans generated by linearizedDP are worse.

Similarly, most of the plans obtained by linearizedDP are optimal or near optimal for the generated queries with known optimal solution. For the larger queries, where the exact algorithms suffer timeouts and thus no optimal solution is known, linearizedDP is mostly able to provide the best solution compared to the plans constructed by the other algorithms. For only for 5% of the queries, other algorithms lead to better solutions than linearizedDP.

Another interesting question is, whether IKKBZ indeed gives a good search-space linearization. To investigate this, we have generated 1,000 random orderings for all queries and ran linearized DP on them. In the following, we compare the best resulting plans with the plans obtained from linearized DP based on the IKKBZ linearization: 82% of the plans using the IKKBZ linearization were

Table 4: Speedups achieved using different BitSet implementations compared to the sparse BitSet

Implementation	min	median	max
64 bit	3.99	10.3	33.0
128 bit	2.09	6.94	28.3
variable	1.14	2.14	5.36

better than any other by a factor of up to 20,000. 17.8% of the plans were more than twice as good, 5.2% more than a factor of 10 better and 0.8% were more than 100 times better. For 10% of the queries, a plan using a different linearization was better, but only up to a factor of 2.3. Thus, IKKBZ is indeed an excellent choice, it is usually much better, and even in the worst cases hardly worse than any of the 1,000 other linearizations.

Performance. LinearizedDP optimizes queries with up to 40 relations within 5ms — a size where DPHyp starts to experience timeouts. Queries with up to 100 relations are optimized within 100ms. And, as can be seen from Figure 10, linearizedDP is able to optimize queries with up to 400 relations within about 20 seconds. It suffers timeouts for 58% of the queries with 500 relations and times out for all queries with 600 or more relations. Note that our adaptive framework already switches to GOO-linDP for queries with more than 100 relations.

6.8 Effect of Implementation Details

Besides the join ordering algorithm itself, using efficient datastructures as described in Section 5 is crucial for the overall performance of join order optimization. In this section we thus take a detailed look into the impact of those implementations details on the performance of various algorithms.

BitSet Implementations. Efficient join ordering requires efficient handling of sets of relations. We have implemented four different flavors of such sets as described in Section 5. Each of those flavors is used for a different range of query sizes. The slowest of our implementations (*sparse*) uses a sorted vector of relations and is more than 3 times faster than using set from the C++ Standard Library (libstdc++). As the set from libstdc++ is not a viable alternative, we measure by which factor the adaptive join ordering framework gets faster when using the various BitSet implementations compared to using the sparse BitSet. Those speedups are reported in Table 4.

The fastest implementation is the fixed size BitSet for up to 64 relations. It is faster by a factor between 3.99 and 33.0 with a median of 10.3, depending on the size of the query, the shape of the query graph and the join ordering algorithm in use. Doubling the size by utilizing two integers incurs a slight overhead. Still such sets give enormous speedups of between 2.09 and 28.3 with median 6.94. For larger queries, we use a vector of integers (presence of a relation is still indicated by a single bit). This variable sized BitSet gives speedups between 14% and a factor of 5.36 with a median of 2.14. The memory requirements of such a set are linear with the query size. We resort to the sparse BitSet for queries with thousands of joins, where the majority of sets is sparsely populated and memory consumption is reduced dramatically when using the sparse BitSet.

Join Lookup. As described in Section 5, we maintain a *join lookup* structure to retrieve join edges in constant time. We measure the speedups achieved utilizing this structure versus implementations that iterate the list of join edges. With increasing query size, the impact of the join lookup on overall performance becomes prominent for both, IKKBZ and GOO (see Figure 13 in the appendix). For IKKBZ, the speedup reaches a factor of 2 for queries with more than 300 relations. Note that using the join lookup IKKBZ can optimize queries with up to 1,000 relations within 60 seconds. Without the join lookup it times out for queries larger than 800 relations. The relative impact of using the join lookup becomes even greater for GOO. Here, the speedup exceeds a factor of 15 for most of the queries of size 3,000. Again, without the join lookup, larger queries suffer timeouts. With the join lookup in place, GOO is able to optimize queries with up to 5,000 relations within at most 17 seconds.


UnionFind. We implemented two versions of GOO, one that utilizes a union-find data structure to maintain the sets of joined relations and one maintaining a list of join edges. The latter modifies the existing edges every time a join is picked by GOO (possibly deleting redundant edges from the list). Union Find pays off especially for large queries, as additional expensive traversals of the complete edge list are avoided. On average, GOO becomes 43% faster when using an efficient union find implementation.

Careful implementation of datastructures and algorithms is thus crucial once queries become reasonably large. Some algorithms even become applicable to larger queries primarily due to the implementation tricks from Section 5.

7 CONCLUSION

In this paper we introduced an adaptive join order optimization framework handling a wide range of queries from small and easy ones up to mega queries with thousands of relations. Through the application of a novel *search space linearization* onto dynamic programming, we are able to generate plans of good quality even when exact optimization becomes prohibitively expensive. If queries become too large to be handled directly by this linearized DP algorithm, we adaptively introduce a greedy step and still benefit from the freedom gained through search space linearization. This lets quality degrade much more gracefully than switching to a completely different algorithm as some optimizers do. Furthermore, we presented crucial implementation techniques necessary to achieve maximum performance. Using all these techniques, our adaptive framework on the one hand maintains optimality when optimizing small queries. On the other hand, it is able to optimize queries on up to 5,000 relations within less than 20 seconds.

A query optimizer must not only be able to handle this wide range of query sizes, but also needs to support queries with non-inner joins which result in a hypergraph. The search space linearization as presented in this paper is unable to handle hypergraphs and thus we have to fall back to normal Iterative DP using DPHyp in this case. We therefore plan to investigate a generalization of this technique to handle non-inner joins as well.

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

REFERENCES

- [1] Renzo Angles, Peter A. Boncz, Josep-Lluís Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martínez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The linked data benchmark council: a graph and RDF industry benchmarking effort. *SIGMOD Record* 43, 1 (2014), 27–31. <https://doi.org/10.1145/2627692.2627697>
- [2] Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. 2010. Polynomial heuristics for query optimization. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. 589–600. <https://doi.org/10.1109/ICDE.2010.5447916>
- [3] 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*. 282–292. <https://doi.org/10.1145/543613.543650>
- [4] Yijou Chen, Richard L. Cole, William J. McKenna, Sergei Perfilov, Aman Sinha, and Eugene Szedenits Jr. 2009. Partial join order optimization in the paracel analytic database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 905–908. <https://doi.org/10.1145/1559845.1559945>
- [5] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*. 54–67. https://doi.org/10.1007/3-540-58907-4_6
- [6] 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*. 785–796. <https://doi.org/10.1145/1247480.1247567>
- [7] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, and Eric Simon. 2009. 1, 000 Tables Inside the From. *PVLDB* 2, 2 (2009), 1450–1461. <http://www.vldb.org/pvldb/vol2/vldb09-1077.pdf>
- [8] 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*. 726–735. <https://doi.org/10.1007/BFb0054528>
- [9] Pit Fender and Guido Moerkotte. 2013. Counter Strike: Generic Top-Down Join Enumeration for Hypergraphs. *PVLDB* 6, 14 (2013), 1822–1833. <http://www.vldb.org/pvldb/vol6/p1822-fender.pdf>
- [10] Gurobi Optimization, Inc. 2016. Gurobi Optimizer Reference Manual. (2016). <http://www.gurobi.com>
- [11] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. 1997. Seeking the Truth About ad hoc Join Costs. *VLDB J.* 6, 3 (1997), 241–256. <https://doi.org/10.1007/s007780050043>
- [12] R. Hipp et al. 2015. SQLite (Version 3.8.10.2). SQLite Development Team. Available from <https://www.sqlite.org/download.html>. (2015).
- [13] 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>
- [14] 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>
- [15] 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*. 128–137. <http://www.vldb.org/conf/1986/P128.PDF>
- [16] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.
- [17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. <http://www.vldb.org/pvldb/vol9/p204-leis.pdf>
- [18] Guy M Lohman. 2014. Is query optimization a “solved” problem. In *Proc. Workshop on Database Query Optimization*. Oregon Graduate Center Comp. Sci. Tech. Rep, 13.
- [19] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. 545–563.
- [20] Guido Moerkotte, Pit Fender, and Marius Eich. 2013. On the correct and complete enumeration of the core search space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 493–504. <https://doi.org/10.1145/2463676.2465314>
- [21] 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*. 930–941. <http://dl.acm.org/citation.cfm?id=1164207>
- [22] 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*. 539–552. <https://doi.org/10.1145/1376616.1376672>
- [23] 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*. 403–414. <https://doi.org/10.1145/1559845.1559889>
- [24] 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, May 30 - June 1*. 23–34. <https://doi.org/10.1145/582095.582099>
- [25] Anil Shanbhag and S. Sudarshan. 2014. Optimizing Join Enumeration in Transformation-based Query Optimizers. *PVLDB* 7, 12 (2014), 1243–1254. <http://www.vldb.org/pvldb/vol7/p1243-shanbhag.pdf>
- [26] 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>
- [27] Arun N. Swami. 1989. Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*. 367–376. <https://doi.org/10.1145/67544.66961>
- [28] Transaction Processing Performance Council 2017. *TPC Benchmark DS*. Transaction Processing Performance Council. <http://www.tpc.org/>
- [29] Transaction Processing Performance Council 2017. *TPC Benchmark H*. Transaction Processing Performance Council. <http://www.tpc.org/>
- [30] 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*. 1025–1040. <https://doi.org/10.1145/3035918.3064039>
- [31] Florian Waas and Arjan Pellenkoft. 2000. Join Order Selection - Good Enough Is Easy. In *Advances in Databases, 17th British National Conference on Databases, BNCOD 17, Exeter, UK, July 3-5, 2000, Proceedings*. 51–67. https://doi.org/10.1007/3-540-45033-5_5

A APPENDIX

A.1 Detailed Distribution of Optimization Times

In Section 6 we showed the median optimization times for various algorithms. On the following pages we now give a more detailed view onto the distribution of those optimization times within the different size classes. In Tables 5, 6, 7, 8 and 9 we report minimum, 5th and 25th percentile, median, 75th and 95th percentile, maximum as well as the average and standard deviation of the optimization times for the generated tree queries with sizes of 10, 40, 100, 1000 and 5000.

A.2 Effect of the Join Lookup Table

In Figure 13 we show the median speedups achieved by utilizing the join lookup table for GOO and IKKBZ. Again the error bars span from the minimum to maximum speedup.

A.3 Plan Quality using different Cost Models

In Tables 10 and 11, we give the distribution of the relative plan costs for the two additional cost models we investigated.

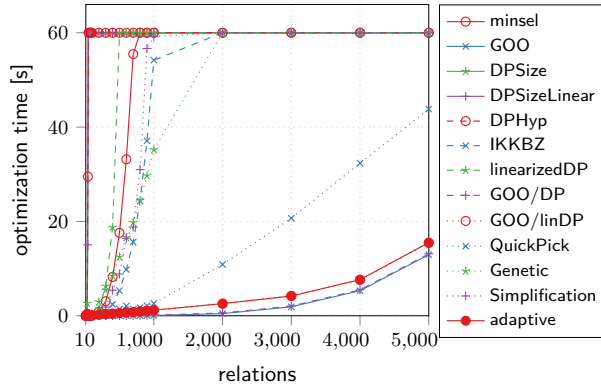


Figure 12: Median Optimization Time for Random Tree Queries of Sizes 10–5000 (100 queries per size)

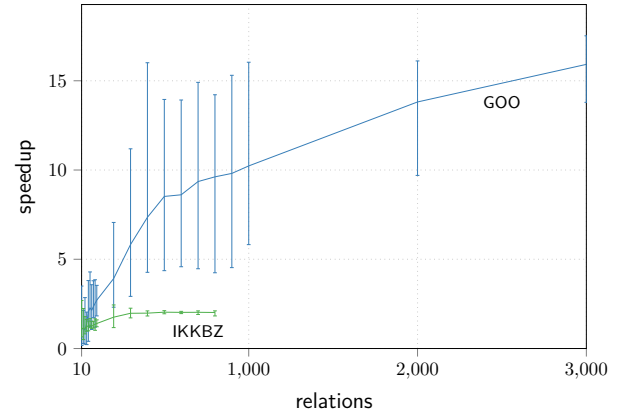


Figure 13: Speedups of GOO and IKKBZ achieved by utilizing the *join lookup* table

Table 5: Distribution of Optimization Times (ms) for Random Tree Queries of Size 10 (100 queries; 60 seconds timeout)

algorithm	min	5%	25%	median	75%	95%	max	avg.	std. dev.
minsel	0	0	0	0	0	0	0	0	0
GOO	0	0	0	0	0	0	0	0	0
DPSize	0	0	0	0	0	0	0	0	0
DPSizeLinear	0	0	0	0	0	0	0	0	0
DPHyp	0	0	0	0	0	0	0	0	0
IKKBZ	0	0	0	0	0	0	0	0	0
linearizedDP	0	0	0	0	0	0	0	0	0
GOO/DP	0	0	0	0	0	0	0	0	0
GOO/linDP	0	0	0	0	0	0	0	0	0
QuickPick	0	1	1	1	1	1	2	1.03	0.223
Genetic	17	18	19	21	23	29.1	43	21.9	4.03
Simplification	0	0	0	0	0	0	0	0	0
adaptive	0	0	0	0	0	0	0	0	0

Table 6: Distribution of Optimization Times (ms) for Random Tree Queries of Size 40 (100 queries; 60 seconds timeout)

algorithm	min	5%	25%	median	75%	95%	max	avg.	std. dev.
minsel	1	1	1	1	1	1	2	1.02	0.141
GOO	0	0	0	0	0	0	0	0	0
DPSize	7,620	29,167	60,000	60,000	60,000	60,000	60,000	56,902	10,139
DPSizeLinear	373	1,816	5,316	14,867	35,879	60,000	60,000	23,212	20,772
DPHyp	1,082	3,750	10,734	29,295	60,000	60,000	60,000	33,350	22,135
IKKBZ	1	1	1	1	1	1	1	1	0
linearizedDP	3	3	3	3	4	4.05	5	3.33	0.57
GOO/DP	0	0	0	0	0	0	0	0	0
GOO/linDP	3	3	3	3	4	5	5	3.39	0.695
QuickPick	6	6	7	7	9	10	11	7.85	1.27
Genetic	64	65	70	73	89	97	103	77.6	11
Simplification	12	13	16	18	20	24	30	18.3	3.3
adaptive	4	4	4	5	5.25	6	7	4.78	0.871

Table 7: Distribution of Optimization Times (ms) for Random Tree Queries of Size 100 (100 queries; 60 seconds timeout)

algorithm	min	5%	25%	median	75%	95%	max	avg.	std. dev.
minsel	23	23	23	24	24	24	24	23.7	0.446
GOO	0	0	0	0	0	0	0	0	0
DPSize	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
DPSizeLinear	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
DPHyp	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
IKKBZ	11	11	11	12	12	12	12	11.7	0.465
linearizedDP	67	69	72	74	77	80	98	74.2	4.16
GOO/DP	1	1	1	1	1	1	1	1	0
GOO/linDP	65	68	69	71	73	76	80	71.2	2.91
QuickPick	26	27	27	28	28	29	30	27.8	0.842
Genetic	228	234	242	250	260	276	301	252	13.9
Simplification	40	45	52	57	61	72	84	57.4	8.45
adaptive	70	72	74	76	79	85.1	118	77.8	7.07

Table 8: Distribution of Optimization Times (ms) for Random Tree Queries of Size 1000 (100 queries; 60 seconds timeout)

algorithm	min	5%	25%	median	75%	95%	max	avg.	std. dev.
minsel	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
GOO	65	66	68	70	89	106	115	77.5	13.9
DPSize	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
DPSizeLinear	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
DPHyp	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
IKKBZ	52,040	52,571	53,394	54,175	55,083	56,144	57,544	54,239	1,162
linearizedDP	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
GOO/DP	131	132	135	137	153	176	216	145	17.6
GOO/linDP	1,113	1,135	1,160	1,187	1,207	1,256	1,296	1,188	36.8
QuickPick	2,489	2,532	2,583	2,610	2,648	2,736	2,754	2,622	58.9
Genetic	32,239	33,089	34,404	35,197	36,262	37,173	38,754	35,282	1,323
Simplification	17,568	27,636	45,119	59,149	60,000	60,000	60,000	51,979	11,330
adaptive	1,115	1,138	1,162	1,187	1,216	1,259	1,303	1,192	39.7

Table 9: Distribution of Optimization Times (ms) for Random Tree Queries of Size 5000 (100 queries; 60 seconds timeout)

algorithm	min	5%	25%	median	75%	95%	max	avg.	std. dev.
minsel	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
GOO	9,019	11,182	12,258	12,962	13,587	14,724	16,707	12,928	1,206
DPSize	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
DPSizeLinear	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
DPHyp	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
IKKBZ	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
linearizedDP	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
GOO/DP	10,220	11,519	12,392	13,115	13,827	14,956	16,757	13,167	1,172
GOO/linDP	11,813	13,425	14,359	15,404	16,202	17,534	19,954	15,376	1,444
QuickPick	42,347	42,793	43,368	43,809	44,298	45,054	46,842	43,860	756
Genetic	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
Simplification	60,000	60,000	60,000	60,000	60,000	60,000	60,000	60,000	0
adaptive	11,346	13,609	14,719	15,464	16,508	17,395	19,526	15,479	1,349

Table 10: Relative Costs with “seeking the truth” cost model for Random Tree Queries of Sizes 10-100 (100 queries per size)

number of relations	normalized costs (avg / 95% / max)					
	10	20	30	40	70	100
minsel	4.4 / 20.6 / 120.7	18.0 / 49.7 / 1.2e3	596.7 / 90.5 / 5.6e4	193.7 / 15.3 / 1.8e4	22.3 / 34.1 / 1.7e3	205.2 / 976.9 / 1.0e4
GOO	1.0 / 1.0 / 1.2	1.0 / 1.1 / 1.7	1.1 / 1.1 / 8.5	1.1 / 1.0 / 9.2	1.3 / 1.1 / 28.5	1.0 / 1.0 / 1.2
DPSize	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0		
DPSizeLinear	4.0 / 10.2 / 120.7	15.7 / 12.8 / 1.2e3	433.9 / 1.2 / 4.3e4	156.0 / 2.9 / 1.2e4		
DPHyp	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0		
IKKBZ	4.0 / 10.2 / 120.7	15.7 / 12.8 / 1.2e3	433.9 / 1.2 / 4.3e4	120.4 / 1.8 / 1.2e4	6.5 / 1.0 / 542.9	1.0 / 1.0 / 1.0
linearizedDP	1.0 / 1.0 / 1.2	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.1	1.1 / 1.0 / 9.2	1.3 / 1.0 / 28.6	1.0 / 1.0 / 1.0
GOO/DP	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.1	1.1 / 1.0 / 9.2	1.3 / 1.0 / 28.6	1.0 / 1.0 / 1.0
GOO/linDP	1.0 / 1.0 / 1.2	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.1	1.1 / 1.0 / 9.2	1.3 / 1.0 / 28.6	1.0 / 1.0 / 1.0
QuickPick	1.0 / 1.0 / 1.1	1.0 / 1.1 / 1.5	1.2 / 1.9 / 4.3	1.2 / 1.9 / 3.2	3.6 / 4.6 / 180.2	2.9 / 8.4 / 22.6
Genetic	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.3	1.0 / 1.1 / 1.1	1.0 / 1.0 / 1.1	1.0 / 1.1 / 1.3	1.0 / 1.1 / 1.3
Simplification	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.1	1.0 / 1.1 / 2.8	91.0 / 2.6 / 8.7e3	2.5e4 / 367.7 / 2.4e6	1.4e4 / 3.8e3 / 1.4e8
adaptive	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.1	1.1 / 1.0 / 9.2	1.3 / 1.0 / 28.6	1.0 / 1.0 / 1.0

Table 11: Relative Costs with C_{mm} cost model for Random Tree Queries of Sizes 10-100 (100 queries per size; 25% indexes)

number of relations	normalized costs (avg / 95% / max)					
	10	20	30	40	70	100
minsel	2.4 / 5.0 / 78.4	14.1 / 11.2 / 914.5	193.7 / 91.1 / 1.4e4	14.0 / 35.8 / 926.1	20.4 / 133.1 / 697.5	1.2e3 / 6.5e3 / 5.7e4
GOO	1.0 / 1.4 / 1.9	1.2 / 1.9 / 2.5	1.2 / 1.7 / 2.3	1.1 / 1.5 / 1.7	1.1 / 1.3 / 1.5	1.2 / 1.4 / 2.2
DPSize	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0		
DPSizeLinear	1.4 / 3.0 / 6.7	1.5 / 4.1 / 13.4	1.2 / 1.5 / 9.3	1.1 / 1.6 / 3.1		
DPHyp	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.0		
IKKBZ	1.4 / 3.0 / 6.7	1.5 / 4.1 / 13.4	1.2 / 1.5 / 9.3	1.1 / 1.4 / 3.1	1.0 / 1.1 / 1.4	1.0 / 1.0 / 1.1
linearizedDP	1.0 / 1.0 / 1.3	1.0 / 1.2 / 1.6	1.0 / 1.1 / 1.7	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.0
GOO/DP	1.0 / 1.0 / 1.0	1.0 / 1.3 / 2.5	1.1 / 1.4 / 2.0	1.0 / 1.2 / 1.6	1.0 / 1.1 / 1.2	1.0 / 1.1 / 2.1
GOO/linDP	1.0 / 1.0 / 1.0	1.0 / 1.3 / 1.6	1.0 / 1.1 / 1.7	1.0 / 1.1 / 1.1	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.1
QuickPick	1.0 / 1.0 / 1.1	1.1 / 1.5 / 2.2	2.0 / 6.4 / 12.7	2.2 / 6.4 / 14.0	16.1 / 23.8 / 1.0e3	13.5 / 48.2 / 141.0
Genetic	1.0 / 1.0 / 1.0	1.0 / 1.2 / 2.1	1.0 / 1.3 / 1.7	1.0 / 1.2 / 1.8	1.2 / 1.7 / 2.7	1.3 / 2.1 / 2.9
Simplification	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.4	1.2 / 1.6 / 2.7	10.6 / 2.5 / 926.1	1.5e5 / 327.3 / 1.5e7	1.3e7 / 2.0e4 / 1.4e9
adaptive	1.0 / 1.0 / 1.0	1.0 / 1.0 / 1.2	1.0 / 1.1 / 1.7	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.1	1.0 / 1.0 / 1.0