Query Simplification: Graceful Degradation for Join-Order Optimization

Thomas Neumann
Max-Planck Institute for Informatics
Saarbrücken, Germany
neumann@mpi-inf.mpg.de

ABSTRACT

Join ordering is one of the most important, but also most challenging problems of query optimization. In general finding the optimal join order is NP-hard. Existing dynamic programming algorithms exhibit exponential runtime even for the restricted, but highly relevant class of star joins. Therefore, it is infeasible to find the optimal join order when the query includes a large number of joins. Existing approaches for large queries switch to greedy heuristics or randomized algorithms at some point, which can degrade query execution performance by orders of magnitude.

We propose a new paradigm for optimizing large queries: when a query is too complex to be optimized exactly, we simplify the query's join graph until the optimization problem becomes tractable within a given time budget. During simplification, we apply safe simplifications before more risky ones. This way join ordering problems are solved optimally if possible, and gracefully degrade with increasing query complexity.

This paper presents a general framework for query simplification and a strategy for directing the simplification process. Extensive experiments with different kinds of queries, different join-graph structures, and different cost functions indicate that query simplification is very robust and outperforms previous methods for join-order optimization.

Categories and Subject Descriptors

H.2 [**Systems**]: Query processing

General Terms

Algorithms, Theory

1. INTRODUCTION

Most non-trivial queries combine data from multiple relations, which requires joining them together. Determining the best join order is one of the classical problems of query optimization, as changing the join order can affect query execution times by orders of magnitude. Most handwritten queries join just a few (<15) relations, but in general join queries can become quite large: Some systems like SAP R/3 store their data in thousands of relations [9], and subsequently generate large join queries. Other examples include data warehousing, where a fact table is joined with a large number of dimension tables, forming a star join, and databases that make heavy use of views to simplify query formulation (where the views then implicitly add joins). Existing database management systems have difficulties optimizing very large join queries, falling back to heuristics when they cannot solve them exactly anymore. This is unfortunate, as it does not offer a smooth transition. Ideally, one would optimize a query as much as possible under given time constraints.

When optimizing join queries, the optimal join order is usually determined using some variant of dynamic programming (DP). However finding the optimal join is NP-hard in general, which means that large join queries become intractable at some point. On the other hand, the complexity of the problem depends heavily upon the structure of the query [15], where some queries can be optimized exactly even for a large number of relations while other queries quickly become too difficult. As computing the optimal join order becomes intractable at some point, the standard technique of handling large join queries resorts to some heuristics. Some commercial database systems first try to solve the problem exactly using DP, and then fall back to greedy heuristics when they run out memory. In the literature a wide range of heuristics has been proposed, as we will see in the Related Work section. Most of them integrate some kind of greedy processing in the optimization process, greedily building execution plan fragments that seem plausible. The inherent problem of this approach is that it is quite likely to greedily make a decision that one would regret having more information about the complete execution plan. For example greedily deciding which two relations should be joined first is very hard, as it depends on all other joins involved in the query.

We propose very different approach: If a query is too complex to optimize exactly, we *simplify* it using a greedy heuristic until it becomes tractable using DP. The simplification step does not build execution plans but modifies the join graph of the query to make it more restrictive, ruling out join orders that it considers unlikely. In a way this is the opposite of the standard greedy plan building techniques: *Instead of greedily choosing joins (which is very hard), we choose joins that must be avoided.* The great advantage of

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.

this approach is that we can start with the 'easy' decisions (i.e., the relatively obvious ones) using the heuristic and then leave the hard execution plan building to the DP algorithm once the problem is simplified enough. The resulting optimization algorithm adapts naturally to the query complexity and the given time budget, simplifying the query just as much as needed and then optimizing the simplified query exactly. We can support a wide range of queries with our algorithm, including outer joins, anti joins and semi joins.

Besides presenting a concrete optimization algorithm, we would like to propose this general strategy of query simplification as a new way of optimizing join queries. The greedy component that decides which simplification to perform next is orthogonal to our graph simplification algorithm; while our 'example' heuristic works very well in practice and has some theoretical foundation, other greedy strategies could be developed in the future and plugged into the simplification framework.

The rest of this work is structured as follows: We first give an overview on related work in Section 2, and then examine the impact of the query structure on the runtime of existing algorithms in Section 3. Using these observations, we present an algorithm for simplifying the query graph in Section 4, together with a simplification order heuristic and a theoretical analysis of this heuristics. Finally we present extensive experiments in Section 5, and draw conclusions and discuss open issues in Section 6.

2. RELATED WORK

2.1 Overview

Join ordering is a well studied field, for a survey over different techniques for large queries see for example [19]. Traditionally the optimal join order is computed using dynamic programming. The popular technique of organizing DP by the number of joined relation was introduced by the seminal paper of Selinger et al. [18], and has been adapted to top-down search [5] and many other variants. Different strategies for organizing the DP process are possible, one based upon fast subset enumeration was proposed by Vance and Mayer [21]. However, both of these algorithms exhibit suboptimal runtime. Therefore [13] introduced a DP strategy based upon the structure of the query graph, meeting the lower bound for all enumeration-based DP algorithms. Different variations of this algorithm have been proposed, a top-down formulation using memoization in [2], and a generalization to hypergraph-structured query graphs in [14]. As we use graph-based DP as a building block in our approach, we discuss it in more detail in Section 2.2. Another way to compute the optimal join order is the IK/KBZ family of algorithms [7, 11]. It has the advantage of finding the optimal join order in polynomial time, but unfortunately requires cost functions with ASI property [7] and acyclic query graphs (no support for hypergraphs), which prevents most practical applications.

For heuristical solutions a wide range of techniques has been proposed, the most promiment ones are greedy heuristics like the min-sel heuristic that sorts the joins according to join selectivity. More advanced greedy heuristics include Greedy Operator Ordering [3], which builds bushy join trees bottom-up by adding the cheapest non-redundant join, and the Maximum Value Precedence algorithm [12], which tries to order joins such that the input of subsequent joins is minimized. These greedy heuristics are cheap to compute, but frequently result in join orders with poor execution time. An interesting hybrid between DP and greedy algorithms is Iterative Dynamic Programming [10]. It solves problems up to a given size optimally, then postulates in a greedy step that the best partial solution found so far is optimal, and repeats the process to derive larger solutions. This allows for a trade-off between optimality and optimization time.

Another family of approaches to heuristical join ordering are randomized algorithms. Popular techniques include Simulated Annealing [20] and its variant two-phase-optimization [8], which traverse the search space in a randomized way to find good join trees. Other alternatives include genetic algorithms [19], which are a different way to navigate the search space, and TF [4] and QuickPick [22], which are based upon random sampling of the search space. These algorithms are usually used with some kind of time budget and produce the best join tree found within the budget. They tend to produce better results than greedy heuristics, but the quality of the solution is hard to predict in general.

2.2 Graph-Based Join Ordering

As our query simplification approach is coupled with graph-based join ordering as proposed in [2, 13, 14], we describe it in more detail here. The graph-bases algorithms organize their DP strategy occording the the *query graph*, i.e., the graph $G = (V, E)$ where V is the set of joined relations and an edge $v = (R_i, R_j)$ is in E if there is a join predicate combining R_i and R_j. When building join trees without cross products, the query graph describes all join possibilities induced by the query.

We observe that a set of relations can be joined if and only if the relations form a connected subgraph in the query graph, simply due to the way the query graph is constructed. We can observe further, that, given two connected sets of relations S_1 and S_2, they can be joined together if and only if they are disjoint but have at least one connecting edge in the query graph. These two observations are the basis of graph based DP: We can solve the join ordering problem by enumerating all connected subgraphs S_1 of the query graph, and for each connected subgraph S_1 enumerating all connected subgraphs S_2 in the complement (i.e., the query graph without S_1) that are connected to S_1 [13]. We know that S_1 and S_2 are joinable, and enumerating S_1 and S_2 in a suitable orders allows us to incrementally fill the DP table.

The complexity of the graph based DP algorithms depends on the structure of the query graph: Some can be optimized easily, while others are complex. One can show that the graph-based DP algorithms enumerate the minimal number of join candidates [13], thus one cannot expect to find a better DP strategy. Note that the query graph is a hypergraph in general [14], as join predicates can involve more than two relations. This complicates subgraph enumeration, but the principle remains the same. For queries with non-inner joins, we can encode the reordering restrictions as hyperedges in the query graph [14], which makes graph-based DP a very versatile optimization strategy.

3. ON OPTIMIZING JOIN QUERIES

Optimizing the join order is one of the most important steps of query optimization, as changes in the join order can affect the query execution times by orders of magnitudes. Unfortunately computing the optimal join order is NP-hard

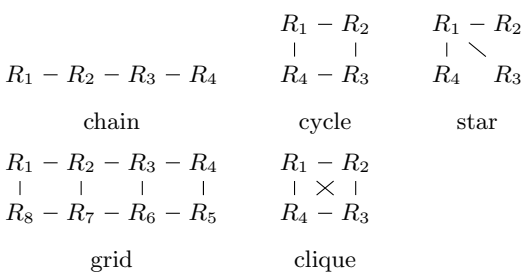

Figure 1: Different Types of Query Graphs

in general [7], and the standard technique of using dynamic programming fails if the query is large enough.

Still, there are large differences in problem complexity even for queries of the same size. When disregarding cross products, the join predicates included in the query induce a *query graph*: The relations accessed in the query form the nodes, and nodes are connected by an edge if there is a join predicate involving the corresponding relations (see Section 2.2). The query graph describes all possible joins for a given query, and indeed it can be shown that the problem complexity of finding the optimal join order depends on the structure of the query graph.

Some exemplary types of query graphs are shown in Figure 1. Clique queries, where there is a join predicate between any two relations involved in the query, are the worst-case scenario for join ordering. Here any combination of relations is joinable, all joins affect each other through redundant join edges, and both the space complexity and the runtime complexity of the best known algorithm increases exponentially [13] (in the order of $O(2^n)$ and $O(4^n)$, where n is the number of relations). For clique queries there is little hope of ever finding a good algorithm, but fortunately large clique queries never occur in practice. Chain queries on the other hand, where relations are joined sequentially, are quite common in practice and much easier to optimize: Any join tree without cross products must only consist of relations that are neighboring in the chain, i.e., that form a subchain. As there are less than n^2 subchains of a chain of length n, and we can join a subchain only with less than n other (neighboring) subchains, we get a space complexity of $O(n^2)$ and a time complexity of $O(n^3)$. Other graph structures are between these two extremes. Star queries, which are common in data warehouse applications where dimension tables are joined to a central fact table, have a space complexity of $O(2^n)$ and a time complexity of $O(n2^n)$. Cycle queries have the same asymptotic complexity as chain queries, but are harder to optimize for some algorithms due to the cycle. Grid queries have been proposed in the literature [19] as hard optimization problems. They are not as extreme as cliques (and thus more realistic), but still hard to optimize due to the large number of cycles.

The practical impact of these complexity differences can be seen in Figure 2. It shows the optimization time using our implementation of the DPhyp algorithm [14] and the setup discussed in Section 5. One observation here is that while small queries (<10 relations) can be optimized quickly regardless of the graph structure, larger queries soon become too expensive for everything except chain and cycle queries. Clique queries are particular bad, of course, but even the data warehousing star queries are too complex relatively soon. For really large queries (e.g., 50 relations), finding the optimal solution using DP is out of test question for most query types.

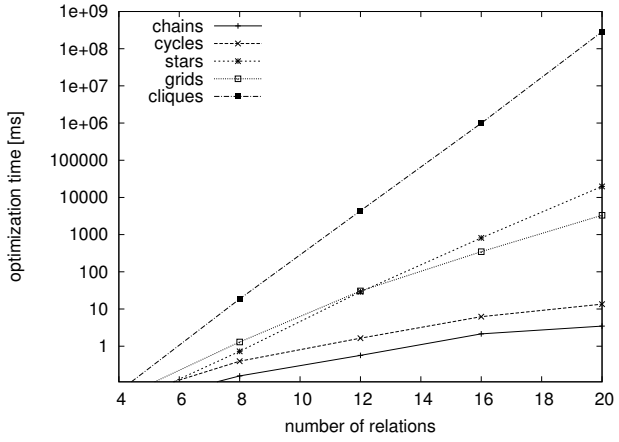

Figure 2: Runtimes for Different Query Graphs

Note that while we use the term *query graphs* (as usually done in the literature), these graphs are really hypergraphs, as pointed out in [14]. Queries can induce non-standard edges that connect sets of relations either by a complex join predicate (for example $R_1.a + R_2.b = R_3.c$), or by using non-inner joins.

4. GRAPH SIMPLIFICATION ALGORITHM

After examining the impact of the query graph structure on optimization time, we now present an algorithm to simplify the query graph as much as needed to allow for a dynamic programming solution. We first discuss the simplification itself, then how this can be used to simplify a query graph as much as needed, and then our edge selection heuristic (which is orthogonal to the main simplification algorithm). Finally we show that our approach is plausible by proving optimality for star queries and certain classes of cost functions.

During this section we assume that the query has been brought into proper query (hyper-)graph form. In particular we assume that all non-inner joins have been expressed as hyperedges, as suggested in [14]. This allows us to reason about graph structures alone without violating proper query semantics.

4.1 Simplifying the Query Graph

When a query graph is too complex to solve exactly, we perform a *simplification step* to reduce its complexity. Note that with *simplification* we mean a simplification of the underlying optimization problem. The graph itself becomes more complex, at least for a human. This is illustrated in Figure 3. The original query is a star query with three satellite relations. The number of possible join trees (ignoring commutativity) is $3! = 6$, as any linear join order is valid. To reduce the search space we look for decisions that are relatively easy. For example if $R_0 \bowtie R_1$ is very selective and $R_0 \bowtie R_2$ greatly increases the cardinality, it is probably a good idea to join R_1 first (for the real criterion see Section 4.3). We thus modify the join $R_0 \bowtie R_2$ into $\{R_0, R_1\} \bowtie R_2$. This describes that we can join a join tree

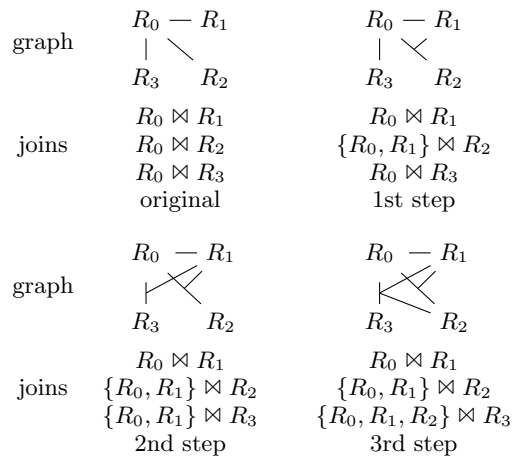

Figure 3: Exemplary Simplification Steps for a Star Query

SimplifyGraph$(G = (V, E))$
$j_1 = \emptyset$, $j_2 = \emptyset$, $M = -\infty$
// Find the most beneficial simplification
for each $S_1^L \bowtie_1 S_1^R \in E$
  for each $S_2^L \bowtie_2 S_2^R \in E$, $\bowtie_1 \neq \bowtie_2$
    // Does $\bowtie_1$ neighbor $\bowtie_2$?
    if $((S_2^L \subseteq S_1^L \vee S_2^R \subseteq S_1^L) \wedge (S_2^L \cup S_2^R \not\subseteq S_1^L)) \vee$
        $((S_2^L \subseteq S_1^R \vee S_2^R \subseteq S_1^R) \wedge (S_2^L \cup S_2^R \not\subseteq S_1^R))$
      $b =$ *orderingBenefit*$(S_1^L \bowtie_1 S_1^R, S_2^L \bowtie_2 S_2^R)$
      if $b > M \wedge (\bowtie_2$ could be ordered before $\bowtie_1)$
        $j_1 = S_1^L \bowtie_1 S_1^R$, $j_2 = S_2^L \bowtie_2 S_2^R$, $M = b$
// No further simplification possible?
if $j_1 = \emptyset$ **return** G
// Order $j_2 = S_2^L \bowtie_2 S_2^R$ before $j_1 = S_1^L \bowtie_1 S_1^R$
if $(S_2^L \subseteq S_1^L \vee S_2^R \subseteq S_1^L) \wedge (S_2^L \cup S_2^R \not\subseteq S_1^L))$
  return $(V, E \setminus \{j_1\} \cup \{(S_1^L \cup S_2^L \cup S_2^R) \bowtie_1 S_1^R\})$
else
  return $(V, E \setminus \{j_1\} \cup \{S_1^L \bowtie_1 (S_1^R \cup S_2^L \cup S_2^R)\})$

Figure 4: Pseudo-Code for a Single Simplification Step

containing R_0 and R_1 with a join tree contain R_2, and forms a hyperedge in the query graph. The search space shrinks to 3 possible trees, as now R_1 is required to come before R_2. R_3 can still be joined arbitrary, either before R_1, between R_1 and R_2, or after R_2. We can reduce the search space to two trees by requiring R_1 to be joined before R_3 (2. step), and finally to just one valid tree by ordering the join with R_2 before the join with R_3 (3. step). At this point the optimization problem is trivial to solve, but the solution could be poor due to the heuristical join ordering. In the actual algorithm we therefore simplify just as much as needed to be able to solve the optimization problem, and we perform these simplification first where we are most certain about the correct join ordering.

The pseudo-code for a single simplification step is shown in Figure 4. It examines all pairs of joins, and checks if they are *neighboring* in the query graph, i.e., they touch via common relations. The condition is somewhat complex, as the query graph contains hyperedges and not just regular join edges. It checks if $\bowtie_2$ could occur in a subtree of $\bowtie_1$ and if $\bowtie_2$ need not come before $\bowtie_1$ (otherwise ordering has no effect). If they are neighboring, we compute the expected benefit of ordering $\bowtie_2$ before $\bowtie_1$. The implementation of *orderingBenefit* is orthogonal to the simplification itself, it should predict how likely it is that $\bowtie_2$ must come before $\bowtie_1$ (see Section 4.3). We restrict ourselves to ordering neighboring joins as it is hard to make useful predictions about arbitrary unrelated joins. Note that through a series of simplification steps the join neighborhoods increase, such that the algorithm can ultimately order all joins if needed. The algorithm remembers the join pair (j_1, j_2) with the maximum estimated benefit, and modifies the query graph such that j_2 must come before j_1. This creates an hyperedge in the query graph, as now j_1 'requires' all relations involved in j_2 to guarantee the ordering, effectively shrinking the search space.

A detail of the pseudo-code not discussed yet is the condition '$\bowtie_2$ could be ordered before $\bowtie_1$' in the first loop. So far we have assumed that it is indeed possible to order $\bowtie_2$ before $\bowtie_1$, but this might not be the case: First, the query might contain non-inner joins, which are not freely reorderable. Second, if the query is cyclic, a series of simplification steps could lead to a contradiction, demanding (transitively) that $\bowtie_1$ must come before $\bowtie_2$ and $\bowtie_2$ before $\bowtie_1$. To avoid this, we build a partial ordering of joins as a directed graph, deriving the initial one from the original query hypergraph and then ordering the joins as indicated by the simplification step. The condition '$\bowtie_2$ could be ordered before $\bowtie_1$' is effectively a check if an edge $\bowtie_2 \rightarrow \bowtie_1$ would create a cycle in our graph. We used a fast online cycle checking algorithm [16] to avoid performance issues for large graphs.

In general the performance of *SimplifyGraph* can be improved significantly by maintaining proper data structures. As we will see in the next section, the algorithm is applied repeatedly to simplify a query graph, thus it pays off to remember already computed results. We therefore materialize all neighbors of a join, and update the neighbors when a join is modified. Further, we remember the estimated benefit for each neighbor, and keep all joins in a priority queue ordered by the maximum benefit they can get from ordering a neighbor. This eliminates the two nested loops in the algorithm, and greatly improves runtime. In performance experiments, we could derive all 4851 simplification steps of a star query with 100 relations in 160ms.

4.2 The Full Algorithm

Our full algorithms works in two phases: First, it repeatedly performs simplification steps until the problem complexity has decreased enough, and then it runs a dynamic programming algorithm to find the optimal solution for the simplified graph. To check if the graph has been simplified enough, we use theoretical results from [13]: The paper proposes the graph-based DPccp algorithm, where the complexity of the dynamic programming algorithm depends on the number of connected subgraphs in the query graph. More precisely, the number of connected subgraphs is identical to the size of the DP table after the optimization is finished. We therefore simplify the query graph until the number of connected subgraphs decreased sufficiently.

Counting the number of connected subgraphs is not that trivial, but we could use the previous results from graph-

```
GraphSimplificationOptimizer(G = (V, E),b)
// Input: A query graph G and a complexity budget b
// Output: The best plan found under the budget b

// Compute all possible simplification steps
Ḡ = a list of query graphs, G' = G
do
   append G' to Ḡ
   G = G', G' =SimplifyGraph(G)
while G ≠ G'
// Use binary search to find the necessary simplifications
l = 0, r = |Ḡ|, v = r, G_b = Ḡ[r − 1]
while l < r
   m = ⌊(l+r)/2⌋
   c =#connected subgraphs in Ḡ[m] (count at most b + 1)
   if c > b
      l = c + 1
   else
      r = c
      if c < v
         v = c, G_b = Ḡ[m]
// Solve the simplified graph
return DPhyp(G_b)
```

Figure 5: The Full Optimization Algorithm

based query optimization: The DPhyp algorithm [14] (a generalization of DPccp to hypergraphs) solves the join ordering problem by enumerating all connected subgraphs of the query graph and joining them with all connected subgraphs that are disjoint from but connected to the first subgraph. By simply eliminating the enumeration of the second subgraph we get an algorithm that produces all connected subgraphs. Note that we do not have to fill a DP table, as we are only interested in the number of connected subgraphs, and we can stop as soon as we have enumerated more than our maximum number of connected subgraphs. In performance experiments enumerating 10,000 subgraphs in a query graph with 100 relations took 5ms. This means that while checking the problem complexity is not that expensive, we cannot afford to check it after each simplifications step, as there may be thousands of simplification steps.

The full algorithm therefore operates as depicted in Figure 5. It is invoked by giving a query graph G and a maximum complexity budget b. It first generates all possible simplifications $\bar{G}$ by applying the *SimplifyGraph* step repeatedly. The complexity of these graphs decreases monotonically, as each simplification step adds more restrictions. Then, it performs a binary search over the list of graphs, and computes the complexity just for the currently examined graph. The graph with the least number of simplification steps that has a complexity $\leq b$ is stored in G_b. Note that G_b could be equal to G, i.e., the original problem, if the graph is simple enough. After the binary search, the optimal solution for G_b is computed by using DPhyp [14].

Again the pseudo-code is simplified. In particular it is not advisable to really materialize all query graphs in $\bar{G}$, as this becomes noticeably expensive for queries with more than 50 relations. Instead, we just remember the two joins (j_1, j_2) selected for merging by the *SimplifyGraph* step. Then we materialize the graphs examined by the binary search by replaying the merge steps based upon these (j_1, j_2) values relative to the last materialized graph. Using these techniques, the full algorithm (including the final *DPhyp* call) takes less than one second for a star query with 50 relations and a complexity budget of $10,000$ in our experiments. Note that we can even avoid generating all possible merge steps: By using search techniques for unbounded search (e.g., [1]) we can generate merging steps as required by the search strategy. This does not change the asymptotic complexity, but it is more efficient if most queries require few or no simplification steps (which is probably the case in practice).

4.3 Join Ordering Criterion

So far we have assumed that the simplification algorithm can somehow estimate the benefit of ordering $\bowtie_2$ before $\bowtie_1$. In principle this is orthogonal to the simplification algorithm, and different kinds of ordering criterion could be used. In our experiments we used the following estimation function, which compares the relative costs of the join orders, and gave very good results:

$$\text{orderingBenefit}(X \bowtie_1 R_1, X \bowtie_2 R_2) = \frac{C((X \bowtie_1 R_1) \bowtie_2 R_2)}{C((X \bowtie_2 R_2) \bowtie_1 R_1)}$$

The rational here is that if joining first R_2 and then R_1 is orders of magnitude cheaper than first joining R_1 and then R_2, it is very likely that the join with R_2 will come before the join with R_1 in the optimal solution, regardless of the other relations involved. As the simplification algorithms orders the highest expected benefit first, it first enforces orderings where the cost differences are particularly large (and thus safe).

Note that the criterion shown above is oversimplified. First, computing the cost function C is not trivial, as we are only comparing joins and do not have complete execution plans yet. In particular information about physical properties of the input is missing, which is required by some cost functions. One way to avoid this is to use the C_{out} cost functions for the benefit estimation, which minimizes the size of intermediate results:

$$C_{out}(T) = \begin{cases} 0 & \text{if } T \text{ is a relation } R_i \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

The advantage of C_{out} is that it can be used without physical information, and further the optimizations based upon C_{out} are usually not that bad, as minimizing intermediate results is a plausible goal. Using the real cost function would be more attractive, but for some cost functions we can only use the real cost function in the final DP phase, as then physical information is available.

The second problem is that we are not really comparing $X \bowtie_1 R_1$ with $X \bowtie_2 R_2$, but $S_1^L \bowtie_1 S_1^R$ with $S_2^L \bowtie_2 S_2^R$, where $\bowtie_1$ and $\bowtie_2$ are neighboring hyperedges in the query graph. The are multiple cases that can occur, here we assume that $S_2^L \subseteq S_1^L$, the other cases are analogous. We define $|S|_\bowtie$ as the output cardinality of joining the relations in S:

$$|S|_\bowtie = (\Pi_{R \in S} |R|) * (\Pi_{\bowtie_i \in V_{|S}} sel(\bowtie_i)).$$

Then the joins $S_1^L \bowtie_1 S_1^R$ and $S_2^L \bowtie_2 S_2^R$ can be interpreted as $X \bowtie_1 R_1$ and $X \bowtie_2 R_2$ with $|X| = \max(|S_1^L|_\bowtie, |S_2^L|_\bowtie)$, $|R_1| = |S_1^R|_\bowtie$, and $|R_2| = |S_2^R|_\bowtie$. Note that we do not have to compute the costs of joining the relations in S_i, as we are

only interested in comparing the relative performance of $\bowtie_1$ and $\bowtie_2$. Note further that the accuracy of the prediction will increase over time, as the S_i grow and at some point contain all relations that will come before a join. Therefore it is important to make the 'safe' orderings early, when the uncertainty is higher, and perform the more unclear orderings later when more is known about the input.

4.4 Theoretical Foundation

The join ordering criterion presented in the previous section is a heuristic, and can lead to suboptimal execution plans. However, we can show that in some cases, in particular for star queries with certain cost functions, we can guarantee the optimality of the reduction.

We define that cost function C is *relative order preserving* if the following condition holds for arbitrary relations $R_{0,\ldots,3}$ and arbitrary joins $\bowtie_{1,2,3}$ with independent join predicates:

$$\begin{aligned} &C(R_0 \bowtie_1 R_1 \bowtie_2 R_2) \quad \geq C(R_0 \bowtie_2 R_2 \bowtie_1 R_1) \\ \Rightarrow &C(R_0 \bowtie_3 R_3 \bowtie_1 R_1 \bowtie_2 R_2) \geq C(R_0 \bowtie_3 R_3 \bowtie_2 R_2 \bowtie_1 R_1) \end{aligned}$$

Or, in other words, the optimal relative ordering of $\bowtie_1$ and $\bowtie_2$ remains unchanged by changing the cardinality of R_0 by a factor of α. This is closely related to the known ASI property of cost functions [7], as it can be shown easily that every ASI cost function is relative order preserving. But relative order preserving is more general than ASI, for example a simple sort-merge-join cost function ($C_{SM}(R_1 \bowtie R_2) = C(R_1) + C(R_2) + |R_1| \log |R_1| + |R_2| \log |R_2|$) does not satisfy the ASI property, but is relative order preserving.

As queries we consider star queries of the form $Q = (V = \{R_0, \ldots, R_n), E = \{R_0 \bowtie_1 R_1, \ldots, R_0 \bowtie_n R_n\})$ (can be guaranteed by renaming relations), and require independence between join predicates and a relative order preserving cost function C. W.l.o.g. we assume that the cost function is symmetric, as we can always construct a symmetric cost function by using $\min(C(R_i \bowtie R_j), C(R_j \bowtie R_i))$. Then, star queries have two distinct properties: First, all query plans are linear, with R_0 involved in the first join. Thus, as our cost function is symmetric, we can restrict ourselves to plans of the form $(R_0 \bowtie R_{\pi(1)}) \ldots \bowtie R_{\pi(n)}$, where $\pi(i)$ defines a permutation of $[1, n]$. Second, given a non-empty join tree T and a relation $R_i \notin T$, $T' = T \bowtie R_i$ is a valid join tree and $|T'| = |T||R_i| \frac{|R_0 \bowtie R_i|}{|R_0||R_i|}$. Thus any (new) relation can be joined to an existing join tree and the selectivity of the join is unaffected by the relations already contained in the tree (due to the independence of join predicates). Note that while this holds for star queries, it does not hold in general. For example, clique queries also allow for an arbitrary join order, but the selectivities are affected by previously joined relations.

Using these observations, we now show the optimality for star queries:

LEMMA 1. *Given a query $Q = (V, E)$, a relative order preserving cost function C and four relations $R_0, R_i, R_j, R_k \in V$ $(i \neq j \neq k \neq 0)$. Then $C(R_0 \bowtie_i R_i \bowtie_j R_j) \geq C(R_0 \bowtie_j R_j \bowtie_i R_i)$ implies $C(R_0 \bowtie_i R_i \bowtie_j R_j \bowtie_k R_k) \geq C(R_0 \bowtie_j R_j \bowtie_i R_i \bowtie_k R_k)$.*

PROOF. Follows directly from the fact that $(R_0 \bowtie_i R_i \bowtie_j R_j) \equiv (R_0 \bowtie_j R_j \bowtie_i R_i)$. The join $\bowtie_k$ gets the same input in both cases, and thus causes the same costs. This lemma holds even for non-star queries and arbitrary (monotonic) cost functions. □

LEMMA 2. *Given a query $Q = (V, E)$, a relative order preserving cost function C and four relations $R_0, R_i, R_j, R_k \in V$ $(i \neq j \neq k \neq 0)$. Then $C(R_0 \bowtie_i R_i \bowtie_j R_j) \geq C(R_0 \bowtie_j R_j \bowtie_i R_i)$ implies $C(R_0 \bowtie_k R_k \bowtie_i R_i \bowtie_j R_j) \geq C(R_0 \bowtie_k R_k \bowtie_j R_j \bowtie_i R_i)$.*

PROOF. Follows from the definition of relative order preserving cost functions. □

COROLLARY 1. *Given a query $Q = (V, E)$, a relative order preserving cost function C, three relations $R_0, R_i, R_j \in V$ $(i \neq j \neq 0)$, and two join sequences S_1, S_2 of relations in V such that $R_0 S_1 \bowtie_i R_i \bowtie_j R_j S_2$ forms a valid join tree. Then $C(R_0 \bowtie_i R_i \bowtie_j R_j) \geq C(R_0 \bowtie_j R_j \bowtie_i R_i)$ implies $C(R_0 S_1 \bowtie_i R_i \bowtie_j R_j S_2) \geq C(R_0 S_1 \bowtie_j R_j \bowtie_i R_i S_2)$.*

PROOF. Follows from Lemma 1 and 2. Both assume nothing about R_k except independence, thus $\bowtie_k R_k$ could be a sequence of joins. □

THEOREM 1. *Given a star query $Q = (V, E)$ and a relative order preserving cost function C. Then for any optimal optimal join tree T and pairs of relations R_i, R_j neighbored in T (i.e., T has the form $R_0 S_1 \bowtie_i R_i \bowtie_j R_j S_2$) the following condition holds: Either $C(R_0 \bowtie_i R_i \bowtie_j R_j) \leq C(R_0 \bowtie_j R_j \bowtie_i R_i)$ or $T' = R_0 S_1 \bowtie_j R_j \bowtie_i R_i S_2$ is optimal, too.*

PROOF. By contradiction. We assume that $C(R_0 \bowtie_i R_i \bowtie_j R_j) > C(R_0 \bowtie_j R_j \bowtie_i R_i)$ and T' is not optimal. By Corollary 1 we can deduce that $C(R_0 \bowtie_i R_i \bowtie_j R_j) > C(R_0 \bowtie_j R_j \bowtie_i R_i) \Rightarrow C(T') = C(R_0 S_1 \bowtie_j R_j \bowtie_i R_i S_2) \leq C(R_0 S_1 \bowtie_i R_i \bowtie_i R_i S_2) = C(T)$. This is a contradiction to the assumption that T' is not optimal □

This theorem is a strong indication that our simplification algorithm is plausible, as we know that one of the optimal solutions will satisfy the ordering constraints used by the algorithm. Unfortunately we were only able to prove the optimality by restricting the cost function some more (perhaps unnecessarily): A cost function C is *fully relative order preserving* if it is relative order preserving and the following condition holds for arbitrary relations $R_{0,\ldots,3}$ and arbitrary joins $\bowtie_{1,2,3}$ with independent join predicates: $C(R_0 \bowtie_1 R_1 \bowtie_2 R_2) \geq C(R_0 \bowtie_2 R_2 \bowtie_1 R_1) \Rightarrow C(R_0 \bowtie_1 R_1 \bowtie_3 R_3 \bowtie_2 R_2) \geq C(R_0 \bowtie_2 R_2 \bowtie_3 R_3 \bowtie_1 R_1)$. Again, this property is satisfied by all ASI cost functions. Using this definition, we can show the optimality as follows.

LEMMA 3. *Given a query $Q = (V, E)$, a fully relative order preserving cost function C, three relations $R_0, R_i, R_j \in V$ $(i \neq j \neq 0)$, and three join sequences S_1, S_2, S_3 of relations in V such that $R_0 S_1 \bowtie_i R_i S_2 \bowtie_j R_j S_3$ forms a valid join tree. Then $C(R_0 \bowtie_i R_i \bowtie_j R_j) \geq C(R_0 \bowtie_j R_j \bowtie_i R_i)$ implies $C(R_0 S_1 \bowtie_i R_i S_2 \bowtie_j R_j S_3) \geq C(R_0 S_1 \bowtie_j R_j S_2 \bowtie_i R_i S_3)$.*

PROOF. Follows from Corollary 1 and the definition of fully relative order preserving cost functions. □

THEOREM 2. *Given a star query $Q = (V, E)$ and a fully relative order preserving cost function C. Applying the* GraphSimplificationOptimizer *algorithm repeatedly leads to the optimal execution plan.*

PROOF. As Q is a star query, any linear join order is valid, thus join ordering is done purely based upon costs. The algorithm repeatedly orders the two joins with the largest

quotient, which is guaranteed to be ≥ 1 due to the lack of join ordering constraints. Lemma 3 shows joins can be ordered relative to each other regardless of other relations, thus if the algorithm orders $\bowtie_i$ before $\bowtie_j$ there exists an optimal solution with $\bowtie_i$ before $\bowtie_j$ (analogue to Theorem 1). The algorithm simplified the graph until the joins are in a total order, which uniquely describes one optimal execution plan. □

Besides star queries, we could probably guarantee optimality for any acyclic query with ASI cost functions by ordering the joins according to ranks [7], but it is not clear that this would be desirable. Real-world cost functions do not satisfy the ASI property and thus no meaningful rank can be computed, while our relative cost based ordering can be applied to any monotonic cost function.

5. EVALUATION

We performed extensive experiments to evaluate the quality of our algorithm for varying query structures, query sizes, join types and cost functions. In the following we first discuss the general setup, and then show results for various kinds of queries. All experiments were run on a Core 2 Duo E8400 with 8 GB memory running Linux 2.6.24, using gcc 4.2.3 as compiler.

5.1 General Setup

We generated queries and optimized them using a variety of algorithms. First, we implemented the standard technique of ordering the joins by minimal selectivity (*min-sel*). The quality of results varies, of course, but this can be considered the baseline for greedy algorithms. We included Greedy Operator Ordering (*GOO*) as an example for a more refined greedy algorithm [3], which can significantly outperform min-sel in some cases. We further included *Iterative DP* as the best previously known algorithm that degrades DP to solve problems [10]. We implemented the variant called IDP_1 in [10], as it outperformed IDP_2 in the experiments. As a randomized algorithm, we implemented Simulated Annealing (*SA*). While in the literature the two-phase-optimization [8] is very popular, where ten rounds of iterative improvement are used to find a start position for SA, we found that using *QuickPick* [22] to quickly generate 100 random join trees and then using the best of them as start position gave better results. Finally, the *GraphSimplification* algorithm is our algorithm as described in Section 4.

To get comparable results, we gave all optimization algorithms a time budget. For our algorithm, we choose a complexity budget of $10,000$, which resulted in optimization times of less than one second (including the simplification step). The other algorithms with variable runtime (Iterative DP and SA) we tuned manually to finish within or with slightly more than one second.

As cost functions we used sort-merge-joins and grace-hash-joins from [6], which are known to be quite accurate. To compare the algorithms, we compute the *scaled costs* as follows: We optimize the query using all algorithms described above, and then compute for each algorithm the scaled costs as the costs divided by the best found solution (which is ideally near to the optimum). In the results we compute the scaled costs for 1,000 random queries, and then show the 5% and 95% quantile for each method as an error bar, and the average runtime as data a point. Some methods have significant outliers, which causes the average costs to go outside the error bar. The 5% and 95% quantile are more robust regarding outliers and show the range of relative performance in 90% of the queries.

Note that the scaled costs are a good indicator for the actual execution times of the query. In particular for sort-merge-joins, which show very predictable disk access patterns, cost functions are very accurate and reliably predict query execution times. The absolute runtimes would be affected by the underlying hardware, but these are mostly constant factors and thus cancel each other out when computing scaled costs.

Note further that we measure only relatively large queries (20 and 50 relations) in our experiments to measure the effect of query simplification. Smaller queries are not that interesting here, as they are already cheap to optimize and our algorithm will simply call the exact DP algorithm in most cases (see Figure 2).

5.2 Random Join Queries

relation size	prob.
10-100	15%
100-1,000	30%
1,000-10,000	25%
10,000-100,000	20%

domain size	prob.
2-10	5%
10-100	50%
100-500	35%
500-1,000	15%

Figure 6: Random Query Generation Parameters Proposed by Steinbrunn et al.

In the first set of experiments, we reproduced the setup from [19]. We created random queries by first creating random relations and adding attributes with random domain sizes using the parameters shown in Figure 6. The query graphs are constructed according to the desired shape (chain, star, etc.) and size, and the selectivity of a join is computed by picking two attributes at random from the two relations and choosing $\frac{1}{\max(dom(A_1),dom(A_2)}$ as selectivity. This generation approach closely follows the description from [19], but caused some problems for us for large queries with a high number of edges (in particular grid queries): There it could happen that the result cardinality was extremely small ($< 10^{-6}$ 'tuples') or, more rarely, very large. To avoid these problems we uniformly multiplied all join selectivities by a correction factor α chosen such that the output cardinality stays within 10^6 and 10^9 if necessary. We avoid these problems by integrating key/foreign-key joins in Section 5.3. We used the sort-merge-join cost function here, as the costs of a sort-merge-join can be predicted very well and the scaled costs thus give a reasonable indication of the expected query execution time differences. We included experiments with hash joins below.

For each query type and each query size we generated 1,000 random queries and computed the average scaled costs and the 5% quantile to 95% quantile range as explained in Section 5.1. The results are shown in Figures 7-10. The first observation is that the quality of the execution plans can vary greatly between queries, and that for most algorithms the scaled costs vary greatly (note the logarithmic scale). The min-sel heuristic usually performs worst, outperform GOO only for grid queries. GOO and Iterative DP perform relatively similar, with Iterative DP usually performing slightly better. But both algorithms frequently pro-

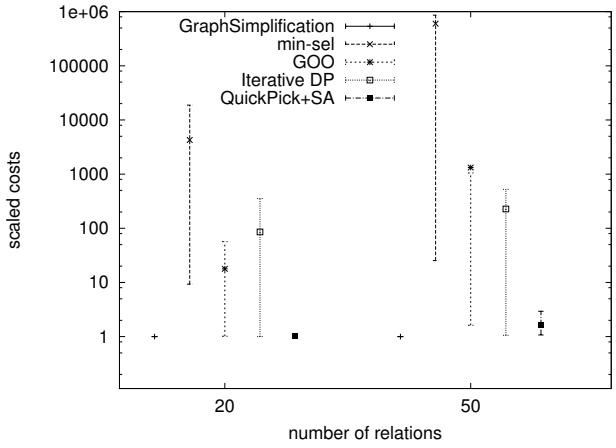

Figure 7: Scaled Costs for 1,000 Random Chain Queries

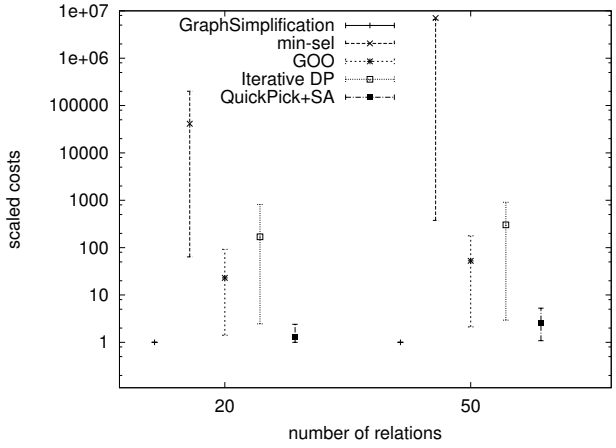

Figure 8: Scaled Costs for 1,000 Random Cycle Queries

duce plans that are a factor of 100 or more worse than the best plan. QuickPick+SA is far more robust than these algorithms, as it tends to produce plans that are within a factor of 10 of the best plan. We found that the initial 100 rounds of QuickPick are crucial for this robustness, as they produce a very good start position. Reducing them greatly increased the variety of costs. Still, although robust against cost extremes, the plans produced by QuickPick+SA are clearly not optimal, with average scaled costs (i.e., a slowdown) of more than 2.5 for cycle queries and more than 62 for grid queries. Our GraphSimplification on the other hand performs consistently very well. For all queries except grid queries it always has scaled costs of 1, which means that it always found the best plan. Grid queries are problematic, as they contain a large number of cycles and our heuristic from Section 4.3 currently ignores the effects of cycles. As can be seen from the large scaled costs of the other approaches, grid queries are difficult to optimize, and they were in fact proposed as hard optimization problems. Still we outperform the other approaches even this cases, performing a factor of 16 better than QuickPick+SA on average for queries with 50 relations.

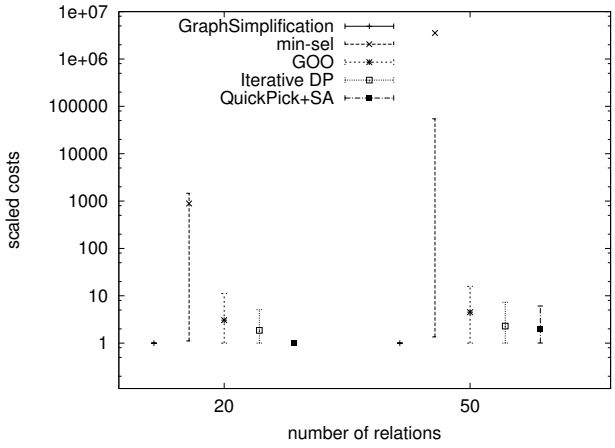

Figure 9: Scaled Costs for 1,000 Random Star Queries

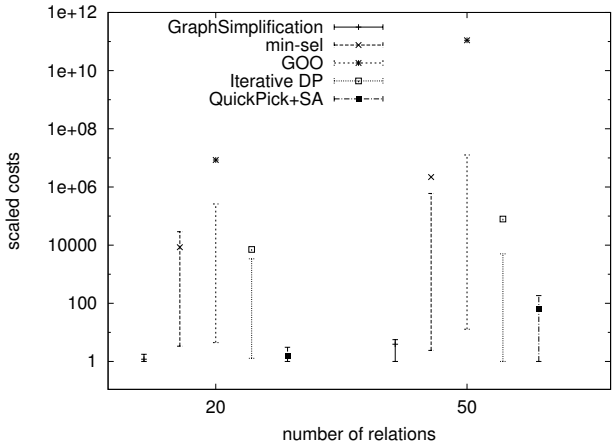

Figure 10: Scaled Costs for 1,000 Random Grid Queries

5.3 Foreign-Key Join Query

We used the query generation from [19] in our experiments as they had also studied optimization of relatively large queries. But during our experiments we noticed that some of the queries seem unrealistic, producing tiny intermediate results (fractions of a tuple), that are then scaled up again by subsequent joins. The tiny intermediate results are not easily avoidable, as using less selective join predicates could result in huge intermediate results. Finding proper values within the given framework seems hard. We eliminated some of these problems by our join scaling mentioned above, but still the queries seemed somewhat arbitrary.

Instead, we propose generating queries with key/foreign key joins. This kind of joins is extremely common in practice, and most joins in large queries will be key/foreign key joins. Further, they have the additional advantage of avoiding the tiny (respectively huge) intermediate result problem, as each tuple from the foreign key side will have at most one join partner, which means that we do not have to worry about exploding intermediate results. We therefore generated queries as follows: We first generated rela-

tions as described in the previous section, but multiplied relation and domain sizes by 100 to get larger data sets to start with. We then generate the query graph and visit the relations in order of decreasing out-degree and decreasing relation size with one class of out-degrees. In each visited relation we examine the neighboring relations, and generate join selectivities as described above with a probability of 10% and foreign-key/key joins with a probability of 90%. For a key/foreign-key join we assuming that the join eliminates a Zipf-distributed part of the key side with $z = 2.0$. With a probability of 10% we visit the target of a such a join, assigned selectivities just as with regular visiting. This creates key/foreign-key chains inside the query graph, which is quite common in practice.

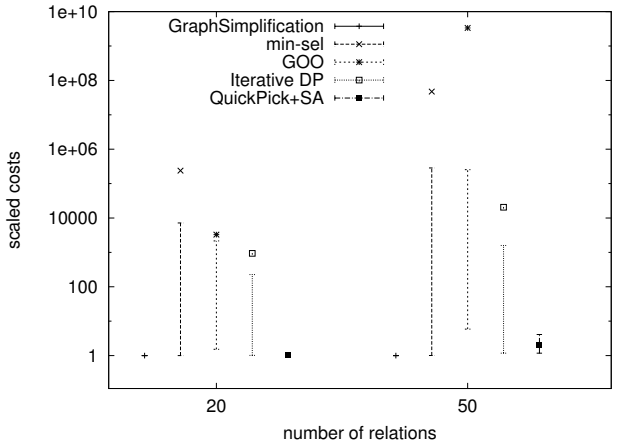

Figure 11: Scaled Costs for 1,000 Random Key/Foreign-Key Chain Queries

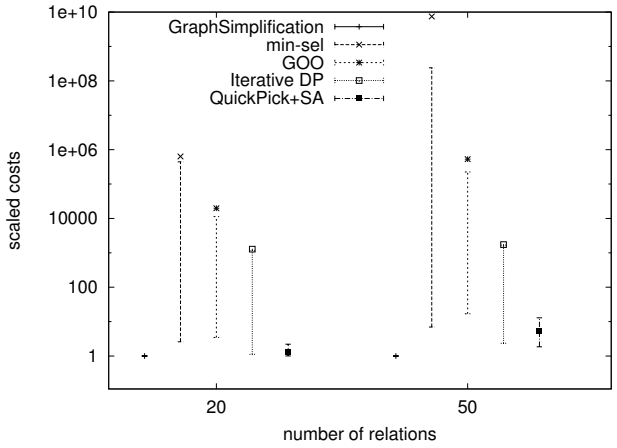

Figure 12: Scaled Costs for 1,000 Random Key/Foreign-Key Cycle Queries

The result are shown in Figure 11-14. The scaled costs are higher than in the first set of experiments, which indicates that these queries are harder to optimize. The relative order of algorithms remains mainly unchanged, only GOO performs worse than in the first experiments. Again our GraphSimplification algorithm has perfect scaled costs of 1 for all but grid queries. Grid queries with 20 relations are

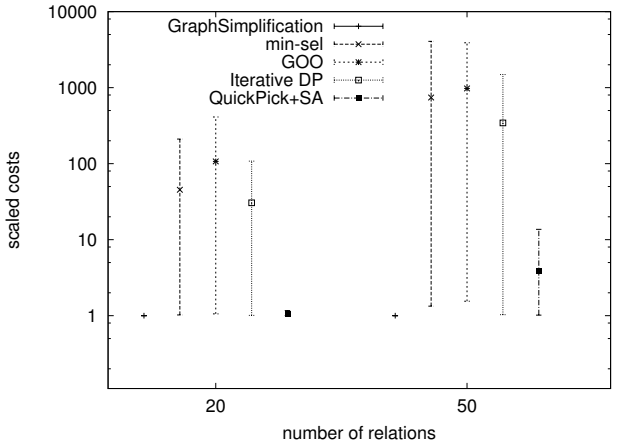

Figure 13: Scaled Costs for 1,000 Random Key/Foreign-Key Star Queries

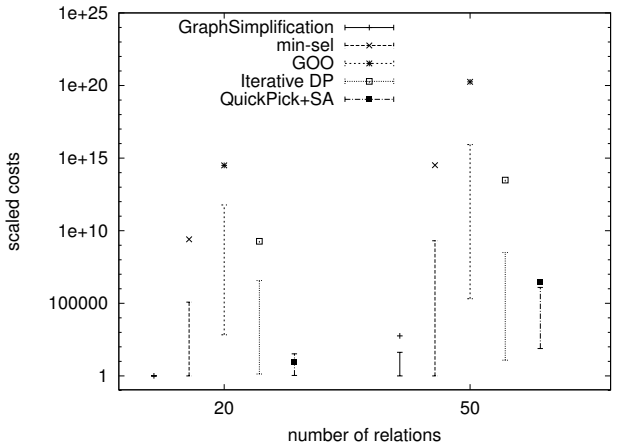

Figure 14: Scaled Costs for 1,000 Random Key/Foreign-Key Grid Queries

fine, as here only a few simplification steps are required, but for grid queries with 50 relations its performance is not that satisfying yet. We will try to improve the estimation of cycle effects in the future. Still, it outperforms the next best algorithm (QuickPick+SA) by more than a factor of $1,000$ on average here, thus the 'bad' behavior of our algorithm is not that bad overall.

5.4 Other Cost Functions

Besides sort-merge-joins, we tried out several other cost functions. The results for grace-hash-joins are shown in Figure 15. Note that GOO 'vanishes' for chain and grid queries as here the scaled costs are outside the logarithmic scale. The cost function from [6] is supposed to be quite accurate, but for very large input it tends to predict extreme costs. We added safeguards against the obvious problems (like division by zero due to rounding issues), but still the scaled costs for GOO tend to be extreme. We are more confident in the prediction value of the sort-merge-join cost function, we therefore used it in the other experiments. Concerning the results, our algorithm performs very well for chain and cycle queries, but has more problems with the other query

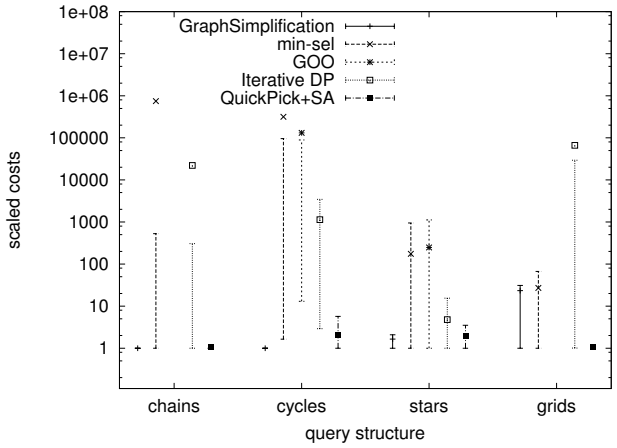

Figure 15: Scaled Costs for 1000 Random Key/Foreign-Key Queries with 50 Relations using Grace-Hash Joins

structures. For star queries we still outperform all other algorithms, but the difference to QuickPick+SA is only 20% on average. For grid queries we actually lose against QuickPick+SA. This has multiple reasons. First, grid queries of size 50 require a lot of reduction steps until they become tractable, which means that we rely heavily upon our ordering heuristic. For grid queries of size 20 we still outperform all other algorithms. Second, as in the previous experiments, our heuristic currently does not predict cycle effects very well. Finally, the cost function is not relative order preserving (see Section 4.4), which further inhibits the heuristic. This is also the reason why we make larger errors for star queries here than in the other experiments.

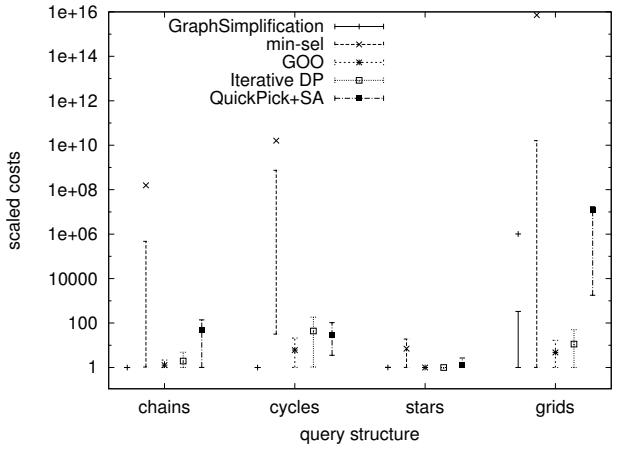

Figure 16: Scaled Costs for 1000 Random Key/Foreign-Key Queries with 50 Relations using C_{out}

For completeness we included the results for minimizing C_{out}, i.e., the size of intermediate results, in Figure 16. This is not a very realistic cost function but has been studied intensively in the literature. In general the results are similar to the previous ones, but some differences are noticeable. First, GOO and in particular Iterative DP perform much better here, at least for star queries (for other query types the absolute scaled costs are similar to previous results). Second, QuickPick+SA performs very poorly, having high costs even in the lower 5% quantile.

In general most heuristics seem to be sensitive to the choice of cost function. GOO for example performs reasonable for sort-merge-joins, very poorly for grace-hash-joins, and quite well when using C_{out}. QuickPick+SA performs quite poorly for C_{out}. Therefore it is hard to predict the quality of these heuristics for different cost functions. To a limited degree the same is true for our algorithm, as the costs for grid queries vary, but for the other, more realistic query graph structures we uniformly outperform all other approaches.

5.5 Non-Inner Joins

While classical (inner) joins are by far the most important type of joins, some queries require non-inner joins like outer joins or antijoins. Our algorithm supports non-inner joins naturally by using the hyper-graph encoding of non-inner joins from [14], no special changes are required. Unfortunately we were unable to perform a meaningful comparison of our algorithm with the other algorithms for non-inner joins, as they had significant issues: Non-inner joins are not freely reorderable, which means that not all syntactically valid join orders are equivalent to the original query. One can detect invalid join orders using EELs [17] or similar techniques [14], but it is unclear how to incorporate these into the other algorithms: All other algorithms except QuickPick+SA contain some kind of greedy step, where a join decision is made greedily that cannot be undone later one. Unfortunately this can result in partial join trees that cannot be completed into complete trees. For example, one algorithms might decide to join the relations R_1 and R_2, although some non-inner join not considered yet requires that R_1 and R_2 must reside in different subtrees of its input. In these cases the greedy algorithms fail to produce a valid solution. It seems to be hard to solve this issue in a greedy strategy, as just checking if placing a join would prevent a complete solution seems to be non-trivial. The QuickPick+SA algorithm can cope with these problems to a certain degree, but it still handles non-inner joins poorly: The initial QuickPick phase has the same issues as the greedy algorithms, frequently failing to build a complete plan. While this is not as disastrous due to the repeated executions of QuickPick, it leads to poor search space sampling and thus to poor initial join trees (in the worst case we only 'find' a canonical join tree). The SA phase, which permutes valid join trees, can also produce invalid join trees. We can ignore these invalid trees, but this severely decreases the effectiveness of SA as most of the time is spend on generating invalid trees. Again, incorporating the reordering constraints into the permutation step seems to be non-trivial without severely biasing the search space exploration.

5.6 Clique Queries

So far we have excluded clique queries from our experiments. Without cross products, clique queries have no practical relevance, as it is hard to imagine a real-world clique query with more than 5 relations. Still, they are interesting from a theoretical point of view, as they are the worst case for join order optimization. In general our algorithm does not support clique queries very well. It is among the best

algorithms for 20 relations, but performs between Iterative DP and GOO for 50 relations. The reason for this is clear: The search space for clique queries is so huge that the algorithm mainly relies upon the heuristic to order joins, and our current heuristic does not handle cycles very well.

An interesting observation here is that min-sel, which usually performs very poorly, outperforms all other algorithm for clique queries. In some preliminary experiments we therefore modified our algorithm as follows: If a query with n relations has more than $2n$ edges, we use the min-sel criterion to select the $2n$ most restrictive joins and treat the others as selections before running the main algorithms. This greatly thins out (and thus simplifies) the query graph for clique queries and drastically improves the resulting execution plans. There are still open issues, but we did not pursue this further as clique queries have no practical relevance.

5.7 The Time/Quality Trade-Off

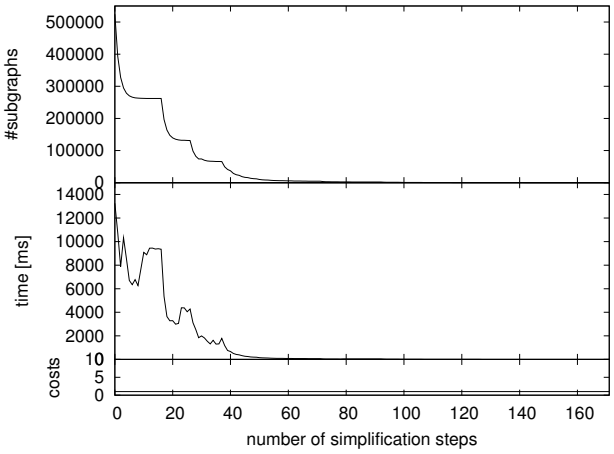

Figure 17: The Effect of Simplification Steps for a Star Query with 20 Relations

After examining the performance of our algorithm in general, we now study the specific effect of simplification on optimization time and query performance. Clearly, each simplification step decreases the search space, i.e., the number of connected subgraphs. Ideally the optimization time goes down analogously, and, unfortunately, the costs will go up if the heuristic makes mistakes. Figure 17 shows how the number of connected subgraphs, the optimization time, and the scaled costs (relative to the optimal solution) change during simplification of a star query with 20 relations. As predicted, the search space shrinks monotonically with simplification. It does not shrink strictly monotonically, as the simplification algorithm sometimes adds restrictions that are already implied through other restrictions, but this is not an issue for the full algorithm due to the binary search. The optimization time follows the search space size, although there are some local peaks. Apparently they are caused by the higher costs of hyperedges for the DPhyp algorithm relative to normal edges. The scaled costs are constantly 1 here, i.e., the algorithm produces the optimal solution regardless of the number of simplification steps. This is due to the theoretical properties of our ordering heuristic (see Section 4.4), which in this case is optimal.

For grid queries the general situation is similar as shown

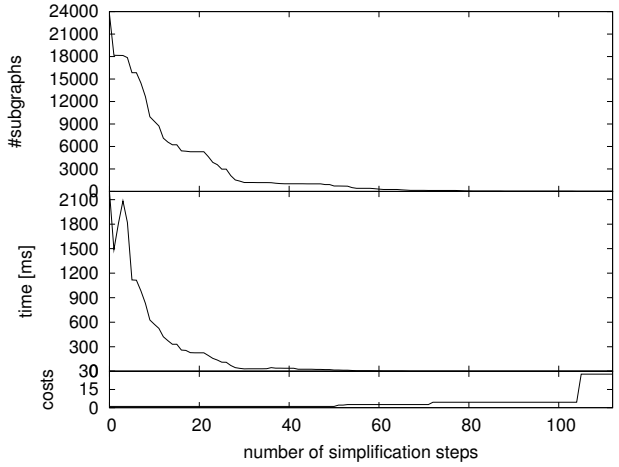

Figure 18: The Effect of Simplification Steps for a Grid Query with 20 Relations

in Figure 18. Search space and optimization time decrease similar to star queries, the costs however increase over time. Initially the heuristic performs only the relatively safe orderings, which do not cause any increases in costs, but at some point it makes a mistake in ordering and causes the costs to increase step-wise. Fortunately this happens when the search space has already been reduced a lot, which means that for simpler queries there is a reasonable hope that the heuristic will never reach the point where it starts making mistakes.

6. CONCLUSION AND FUTURE WORK

Query simplification is a new paradigm for optimizing large join queries. It allows adjusting the quality of the solution to a given optimization budget, and thus preserves optimality if possible given the constraints. We presented a heuristics for guiding the simplification process, which has nice theoretical properties and works very well in practice. Our experiments have shown that these techniques clearly outperform previous approaches, and provide good solutions in all cases.

Future work should include investigating other heuristics for directing the simplification process. In particular cyclic queries could be improved further by more accurately predicting the input cardinalities for joins involved in a cycle. In general it might be useful to use the query graph for more precise cardinality predictions, as it frequently implies that some relations must have to be included in a certain subtree (and thus affect its cardinality). The simplification framework itself is orthogonal to the guiding heuristics, fairly different guiding heuristics could be tried in the future.

7. REFERENCES

[1] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.

[2] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In *SIGMOD Conference*, pages 785–796, 2007.

[3] L. Fegaras. A new heuristic for optimizing large queries. In *DEXA*, pages 726–735, 1998.

[4] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection - why use transformations? In *VLDB*, pages 85–95, 1994.

[5] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[6] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.

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

[8] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD Conference*, pages 312–321, 1990.

[9] A. Kemper, D. Kossmann, and B. Zeller. Performance tuning for SAP R/3. *IEEE Data Eng. Bull.*, 22(2):32–39, 1999.

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

[11] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

[12] C. Lee, C.-S. Shih, and Y.-H. Chen. Optimizing large join queries using a graph-based approach. *IEEE Trans. Knowl. Data Eng.*, 13(2):298–315, 2001.

[13] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006.

[14] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD Conference*, pages 539–552, 2008.

[15] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[16] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):311–337, 2004.

[17] J. Rao, B. G. Lindsay, G. M. Lohman, H. Pirahesh, and D. E. Simmen. Using eels, a practical approach to outerjoin and antijoin reordering. In *ICDE*, pages 585–594, 2001.

[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34, 1979.

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

[20] A. N. Swami and A. Gupta. Optimization of large join queries. In *SIGMOD Conference*, pages 8–17, 1988.

[21] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD*, pages 35–46. ACM Press, 1996.

[22] F. Waas and A. Pellenkoft. Join order selection - good enough is easy. In *BNCOD*, pages 51–67, 2000.