Technische Universität München

Fakultät für Informatik

Lehrstuhl III Datenbanksysteme

Prof. Dr. Thomas Neumann und Bernhard Radke

Seminar thesis

Greedy Operator Ordering

– no subtitle –

Student:	Simon Stolz
Studies:	Informatik: Games Engineering
Semester:	5
Student ID:	03701203
Birth date:	10.5.1999
Address:	Kösching Adalbert-Stifter-Str 29
Phone-No.:	nan
E-Mail:	ge56pid@mytum.de

Munich, 25.11.2019

Table of Contents

List of Figures

List of Tables

Appendices

1 Introduction

In query optimization we usually try to find a join order, that reduces execution cost as much as possible, yet since queries are not always pre-compiled and we might want to optimize and run them on the spot or have to optimize a large number of queries, the time we need to get results from a newly created query does in many cases not only depend on the query execution time but also on the time we need to find an efficient join ordering before starting the execution.

When we look at the beginning stages of databases when there did not already exist huge online mega concerns, a lot of databases will have been used for local inventory management and such. To implement that we do not need a big amount of tables and most of the queries we might want to execute had to join only about 10-15 relations.[1] According to our performance tests, which we introduce later, a Dynamic Programming size (DPsize) implementation optimizing such smaller queries to the optimal solution will take about 5 minutes, which is still fast enough to be efficient for the Databases we are talking about. Yet today the size of databases needed increased immensely due to new kinds of database applications like managing online marketplaces like Amazon or eBay as well as targeted advertising, deep learning and modern social media. As a consequence of that, already in 2010 according to ”Polynomial Heuristics for Query Optimization” the size of databases had increased that much that new queries then had to join 50-100 relations[2], and since online brands continued to grow till today it is likely that we are looking at even lager numbers now. This is a huge problem since the DPsize Algorithm is exponential in time relative to the number of relations joined, this results into DPsize being not suitable for the modern scenarios we described, since it would take too long optimizing a bigger number of said larger queries.[3]

As a result not searching for the best plan but avoiding the bad plans became the new motto according to Leonidas Fegaras. He proposes approaching query optimization by greedily building up join trees instead of cost-based searching and therefore manages to achieve a runtime of O(n^3) with his Heuristic called “Greedy Operator Ordering” short GOO.[1]

With his paper “A new Heuristic for optimizing Large queries” he lays a new foundation for greedy heuristics in query optimization, which later gets generalized into the Enumerate-Rank-Merge (ERM) approach which splits GOO in the three semantic parts Enumerate, Rank and Merge, which makes it easier to extend the greedy approach without having to propose a new heuristic. In “Polynomial Heuristics For Query Op-

timization" ERM was already proposed with several such extensions, which we will discuss in more detail later.[2] After we displayed the theoretical side of both GOO and ERM, we will evaluate their performance by comparing our own ERM implementation against a DPsize contestant.

2 Main Part

2.1 Basic GOO And Cost Function

Now we will look at how GOO greedily builds query execution plans in detail. The general idea of GOO is to best first reduce the size of intermediate results to in the end produce plans that also have low cost while the cost can be approximated by incrementally taking the sum of intermediate results.[1, 2]

The start input for our algorithm is a set of tables T, that contains cardinality information about each table, and an |T| x |T| matrix P that references the set of predicates p(i, j) between each Table i, j element T, p(i, j) having a value of range]0;1] where p(i, j) = 1 indicates that no predicate exists and a cross product is applied between i and j when calculating their new joined carnality. Predicate zero is not included since otherwise our end result would also have cardinality of zero. This would make the whole query invalid because after this predicate it would always return and empty result, therefore we can not use zero to approximate a very low predicate and instead have to use a number greater zero that fits the predicate the best.

At the beginning of our algorithm we create a set of plans P from the input tables T so that for each table t of T there exists one plan only containing t.

Then we determine all possible join options for the current step. For GOO we are considering both bushy trees and cross-products. Bushy trees allow us to have more than one plan that contains more than one table at the same time. An other option would be only considering linear trees which restricts us to only joining new relations onto the root node of the previous joins we made until we joined all relations, while the root node always describes the newly created relation after a hash-join. A join builds a result table (new relation) out of two relations the predicate is used to approximate how many tuples the result contains. The advantage of bushy trees here is that we are considering more possibilities than with linear trees, which results in potentially better solutions and since a bushy-tree can also have the shape of a linear tree the bushy-tree also finds the greedily best solution if it is a linear tree. The downside of considering bushy-trees is additional runtime since we check more options. Considering cross-products means that we are also considering join possibility with tables that have no predicate between them, this again will increase the time to execute GOO but we might find plans that are greedily better. Due to cross-products and bushy-trees we consider every join between any plan of our set of plans except itself as viable join option.[1]

From all the viable options we take the one with the highest rank, which according to our ranking function is the one with the lowest hash-join carnality. We get the cardinality by first

multiplying the cardinality of the two joined relations with each other and then with the join predicate. Once we have determined the best join option we trivially join the two plans by replacing the plan we then would build our hash table from (join base), with a new plan that has the join base as left child and the other relation we hash-joined as right child.

Once we finished merging we delete the right side of the new relation from the set of plans. In the end we only have to update our query graph to fit the merge, more details on that in a later example.

We repeat this starting from finding all valid join options for the current set of plans until only one plan is left, this plan then is our final join tree.[1]

In this Heuristic we have no restrictions on which of the joined plans is supposed to be the join base (also build side), as it has no effect on cardinalities, therefore it will be implementation depended which plans turn out to be in the build side of the hash-join.[2]

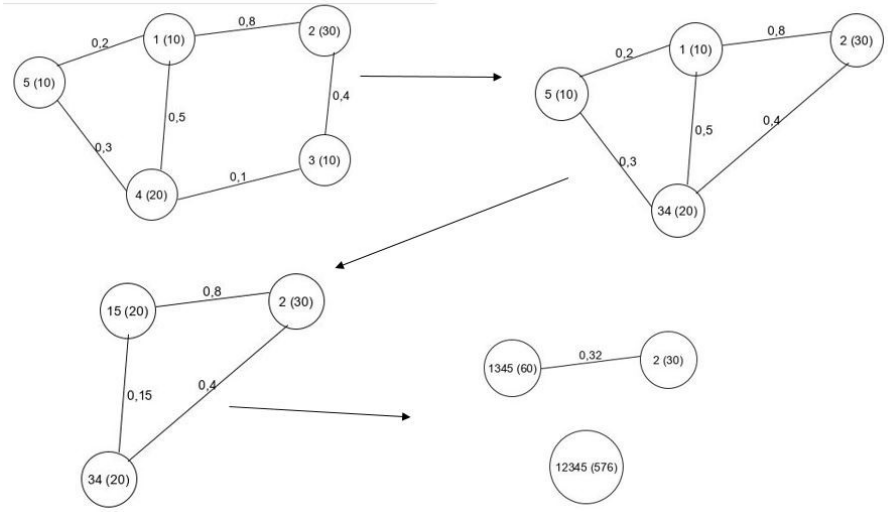

Fig. 1: Goo example

Source: Own creation inspired by Fegeras et al., "A New Heuristic for Optimizing Large Queries", (Arlington: The University of Texas at Arlington, 2010), p 5

Fig. 2: Cost Function shows an example on which we are now applying the algorithm, described before. In the first step joining nodes 3 and 4 has the same estimated cardinality as joining 1 and 5 which shows, that GOO is not necessarily a stable algorithm since we might get different outputs for the same input depending on implementation. We select to join 3 and 4 and merge the two plans into a new one called "34". The new node now has the cardinality 20 which was estimated by our ranking function. After we replaced plan 3 with the new plan we also had to remove plan 4. 34 now has predicates to any relation either plan 3 or plan 4 had one to, if both original plans have predicates to the same relation we have to multiply the two predicates. In the case of 34 we can just copy the predicates p(4,1), p(4,5) and p(3,2) to the new plan since we have no duplicate. In the next step when we select to join 1 and 5 to be cheapest we have the predicates p(5, 34), p(1, 34) and p(1, 2). Here we have to multiply the

predicates p(5, 34) and p(1, 34) to gain the predicate p(15, 34), p(1,2) can simply be copied to p(15, 2). For the following steps we proceed the same until we have joined all plans into one, then we have generated our result join order.

Since we do not optimize the cost directly but the cardinality of intermediate results, the bottom-up greedy Heuristic GOO is not optimal relative to the cost function, to understand what the cost function means for the algorithm we will first explain what a cost function does and introduce the one we are referencing after.
A cost function is used to approximate the execution cost of a query. Cost-guided approaches use this function to determine the best plan. Since the cost function is their ranking function, they usually find the cheapest plan according to that function. With GOO we also aim to archive a solution close to that cheapest plan but use intermediate cardinality as ranking function, after we executed a greedily best join we have no chance to revert it if there would be a better solution after our cost function once we joined more relations.
That means with GOO we are not enumerating all possible join orders but use GOOs ranking function to apply the intermediately highest rank join until all tables are joined, therefore we do not consider every possible join tree and also might not be able to find the cheapest join plan according to our cost function.[1]

$$C_{\text{mm}}(T) = \begin{cases} \tau \cdot |R| & \text{if } T = R \vee T = \sigma(R) \\ |T| + C_{\text{mm}}(T_1) + C_{\text{mm}}(T_2) & \text{if } T = T_1 \bowtie^{\text{HJ}} T_2 \\ C_{\text{mm}}(T_1) + & \text{if } T = T_1 \bowtie^{\text{INL}} T_2, \\ \quad \lambda \cdot |T_1| \cdot \max(\frac{|T_1 \bowtie R|}{|T_1|}, 1) & (T_2 = R \vee T_2 = \sigma(R)) \end{cases}$$

Fig. 2: Cost Function
Source: Victor Leis et al.,"How Good Are Query Optimizers, Really?"(Published in PVLDB 2015),p212

In Fig. 1: Goo example you can see the cost function we are talking about, it is calculated incrementally for each node of the join tree. It describes how we calculate the sum of the intermediate results and also considers index-joins.[4]

2.2 Enumerate-Rank-Merge

The ERM generalizes GOOs greedy agenda by structuring it into the three stages enumerate, rank and merge, which has the advantage that changing restrictions on viable join options, the ranking function and the way we merge the greedily best join does not require us to think of a new heuristic from scratch. The runtime complexity of ERM is O(N^2*R+N*M), which in general is the same as GOO with O(n^3) but since we applied the ERM heuristic was applied here we differentiate between the steps Rank and Merge as follows. N describes the amount of plans we have to join, R is the amount of operations we need to find the highest ranked join and M is the amount of steps needed to merge the two plans we selected to be the highest ranked to join.

Thanks to the ERM subdivision we can now easier analyse the runtime in more detail since we have the runtime of the rank and merge step as own variable.[2]

Additionally to the straight up generalization of the existing GOO, in “Polynomial heuristics for query optimization” the ERM heuristic also comes with several extensions. In this paper we will be discussing the ones called Push, Pull, Switch-HJ and Switch-Idx in more detail, they are used to improve the quality of plans GOO delivers. Push, Pull and Switch-Idx also introduce the consideration of indices that means they provide the GOO heuristic with a way to consider index-joins.[2]

2.2.1 Enumerate and Ranking Step

In the enumerate step we determine which joins are eligible in each iteration of finding the greedily best join determined by a valid function. Since the ERM approach only generalizes GOO we already discussed cross products with bushy-trees in the GOO introduction, we will consider them both in the ERM approach as well. The valid function in the ERM approach takes the place of what we described as finding all viable join options in the GOO introduction. [2.1]

In the ranking step we run a ranking function on each pair of plans, that passed the valid function and hand off the highest ranked pair to the merging step. The ranking function in theory is completely variable, since changes to the ranking function does not bother the rest of the ERM heuristic. For example, on the one hand we could use the ranking function used in GOO, on the other hand we can also change or extend it and for example consider row sizes as well, by integrating them into the function, which can increase the quality of plans for real time benchmarks that determine the execution time of plans[2].

2.2.2 Merge Step

In the merge step we combine the highest ranked plans. For the basic GOO approach we always take the parent nodes of the left and right side and make them the left and right child of a new parent node respectively. The new parent node also contains the newly calculated cardinality and if we calculate it while merging, which we will need for several of the merging extensions, also the cost for the new parent node.[2]

2.2.3 Switch-HJ And Switch-Idx

Switch-HJ applies only for hash-joins, it makes the smaller relation the build side of the join between the two plans. That means if we first wanted to make the larger cardinality plan our left child of the new created plan, also called build side, we swap both plans and make the smaller cardinality plan the build side. This adoption has no impact on the plans cardinality or cost according to our cost function but is advantageous in reality.[2]

Switch-Idx is a more complex extension, it is used to integrate the consideration of indexes into our algorithm, this is important because in many cases applying an index-join might be cheaper than doing a hash-join and GOO is only considering hash-joins so far. Since the cost-function we are using also considers index-joins this extension also has potential to increase the quality of plans according to our cost-function.

Switch-Idx can in general be applied whenever an index-join is cheaper to execute than a hash-join alternative. In this case we would have to turn said join into an index-join and swap the joined plans so that that build side is the Plan that contains more than one table. It is important to know thought that an index-join can only be applied when one of the plans contains only a single table and an index between the two plans exists.

Switch-Idx can be either directly integrated into the ranking function or only applied on the two nodes we want to merge via checking, if applying an index-join would decrease the cost of the tree in the merging step.[2]

2.2.4 Push

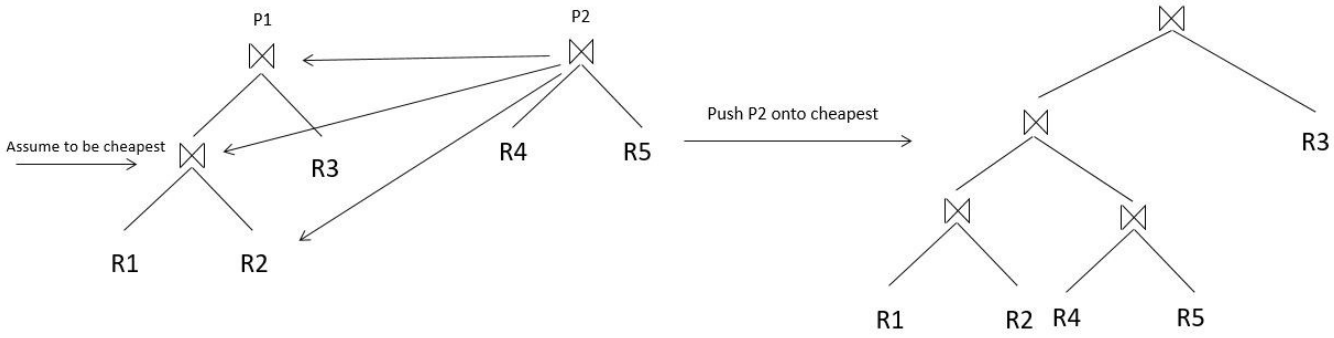

Fig. 3: Push
Source: Own creation

Since we are the greedy technique is not optimal we might make mistakes building our tree. For example if we have the three relation A, B and C to join and "(A join B) join C" would be cheaper than "(A join C) join B" after our cost function but joining A on C is better after our ranking function than joining A on B. Then after the ERM approach, we have to first join A on C, but when we would be able to join B in the next step we can not build the cheapest tree without the Push extension[2].

The push extension focuses on correcting these early mistakes by pushing one plan into the other if it decreases the cost of the root node. Pushing means that instead of simply merging both plans by merging them at their root we consider taking the root of one plan and merge it with a sub-tree or leaf of the other plan, the root of the merge then takes the place of the tree-node that was pushed onto. For pushing we do not include cross products as valid pushes since pushing the one tree into the other with a predicate of 1 will only increase the cost of the tree or keep it the same in the best case, when we push it down the tree. If we considered cross-products in the rest of the algorithm we have to watch out for the special case that there is no node we can push our tree onto that has a predicate lower than 1.

We can try to push each tree into the other and we only consider pushes for which a predicate exists, once we determined all pushes that meet these restrictions, we need to calculate the cost for each push alternative. For each push alternative we only have to update cost and cardinalities for nodes of higher level relative to the node we push onto. All this can be done in linear time per push alternative, if we assume all our predicates to be independent.[2]

On the left hand side of Fig. 3: Push we want to push the right plan into the one on the left. Assuming we only have a predicate between R2 and R4 we have three options to do that, as shown in the graphic. The combined tree on the right depicts how a completed merge looks like under the assumption of pushing the right tree onto the merge node of R1 and R2 result-

ing in the cheapest cost root node. Switch-Idx and Switch-HJ can both be used with this extension[2]

2.2.5 Pull

Since we always execute the greedily best join, there is the possibility that after we add a new plan, for the new root there arise additional possibilities for index-joins, with the root as left -side, involving tables that are already part of a hash-join lower down in the tree as right side of the index-join.

With the pull technique we try to make these index-joins possible although we already made a bad decision further down in the tree, it is executed after we have our base merge completed. Once a new plan was joined on our base there might be new ways to create index joins by trying to pull any table that is not already the right side of an index-join onto the root of the plan as the inner side of a new index-join. The one pull option that has the lowest cost will be executed, if no pull option is better than the already existing tree, no pull is executed. Calculating each pull option can be done in constant time under the same conditions mentioned in push.[2]

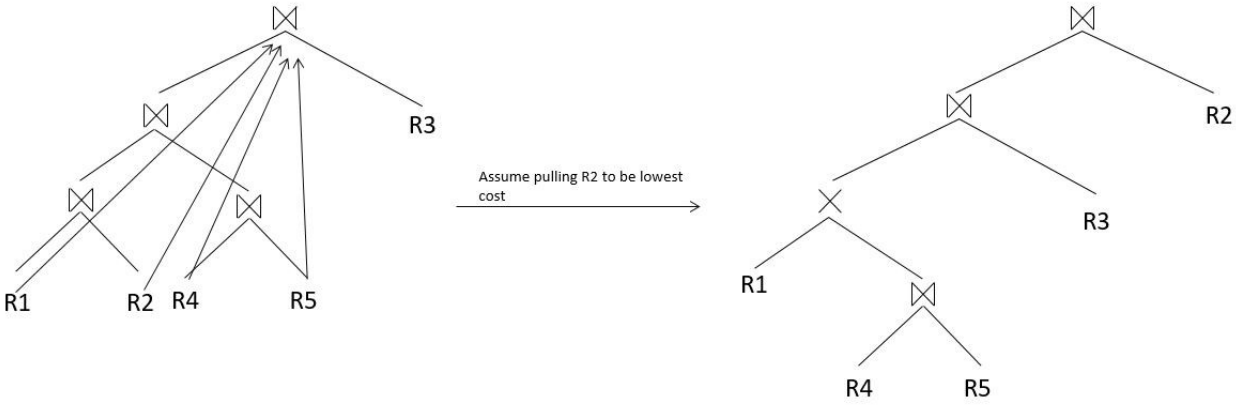

Fig. 4: Pull
Source: Own creation

In Fig. 4: Pull a pull scenario is illustrated, on the left we can see our newly joined plan and all the pull options we have. For the example we assume that pulling R2 onto the root of the tree is the cheapest option. On the right hand side of the illustration you see how the pulled tree would look like. There the parent node of R1 gets turned into a cross-product in the merged tree, that can happen if the only predicate between the sub tree of R1 and R2 and the sub tree of R4 and R5 is between R2 and R4, and R2 gets pulled up onto the root.

2.2.6 Extension Complexity

```
BSizePP (T: set of tables, J: join graph)
01 P = { Scan(t) | t ∈ T }
02 while |P| ≥ 1
03    E = { (P1, P2) ∈ P × P }    // consider bushy trees
04    find (P1, P2) ∈ E minimizing minSize(P1, P2)
05    Pn = best of Push(P1, P2) and Pull(P1, P2)
            // Includes Switch-HJ and Switch-Idx
06    P = P − {P1, P2} ∪ {Pn}
07 return p ∈ P
```

Fig. 5: ERM Pseudocode
Source: Nicolas Bruno et al."Polynomial Heuristics for Query Optimization"(USA: Microsoft Corp.), p 593

In Fig. 5: ERM Pseudocode you can see how the extensions we talked about are integrated into the ERM approach, as we yet mentioned it has a runtime of O(N^2*R+N*M), R itself has the complexity O(N^2), which on the one hand means that ERM does not have a better complexity than GOO and on the other hand that M can have a complexity of O(N^2) without increasing the complexity of the whole algorithm. The basic merging approach used in GOO is linear in time, that allows us to add the merging the four merging extensions without increasing the complexity. Since the Switch-Idx extension can also be applied in the ranking function in this case we would have to watch out that our integration of indices into the ranking function is possible to be calculated in linear time. [2]

2.3 Performance Evaluation

We now have discussed why GOO was proposed and what the advantages and downsides of GOO and its generalization ERM should be in theory, now we will put what we have learned into application and evaluate how these strengths and weaknesses apply by evaluating its performance against a DPsize competitor

Again the general advantage of the GOO and ERM approach is, that we find plans of good quality in O(n^3) and don´t need exponential time to find a solution like the DP algorithm, but we trade off the quality of plans for execution time, since we don´t necessarily find the optimal plan.[1]

2.3.1 Quality Of Plans

All the samples where executed on an Ubuntu virtual machine.

For evaluation we implemented the simple GOO algorithm considering cross-products and bushy-trees. We first want to take a look at the quality of our generated plans. To evaluate the quality we set the execution cost of results from our GOO implementation and a basic DP

contestant, that we did not implement ourselves, in relation. We could do that either by executing the queries on a benchmark or use a cost approximation function. In our case we use the cost function from Fig. 2: Cost Function to compare the quality of results of both Algorithms. Here it is important to note that also our cost function is only an approximation of the actual cost for executing a query, therefore our evaluation as well is only an approximation for a real query execution benchmark.

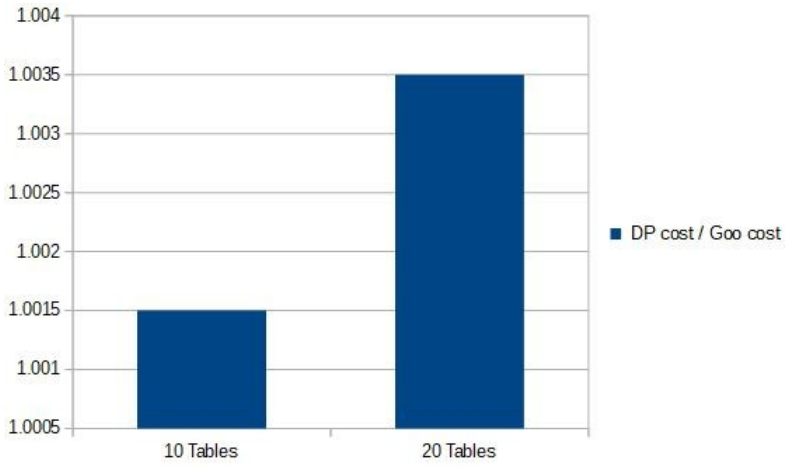

Fig. 6: Goo inaccuracy for realistic data
Source: Own creation

We first used a set of queries that where created to simulate real life databases, these had the sizes of 10 and 20 and we executed GOO and DP for 99 of each size to generate the graph in Fig. 6: Goo inaccuracy for realistic data. The queries used are the same as the ones used in "Adaptive optimization of very large join queries.".[5] Our graph sets the costs of the two executions in relation by dividing the cost of the DP result by the one of GOO. For the evaluation diagrams Fig. 6: Goo inaccuracy for realistic data, Fig. 7: Goo inaccuracy for naive random queries, Fig. 9: GOO vs. DP runtime in seconds, Fig. 10: GOO runtime in milliseconds and Error: Reference source not found we always used the average runtime of all executions as optimization time reference. This first measurement shows that for increasing the size of the relations comes decreasing quality of the GOO approximation.

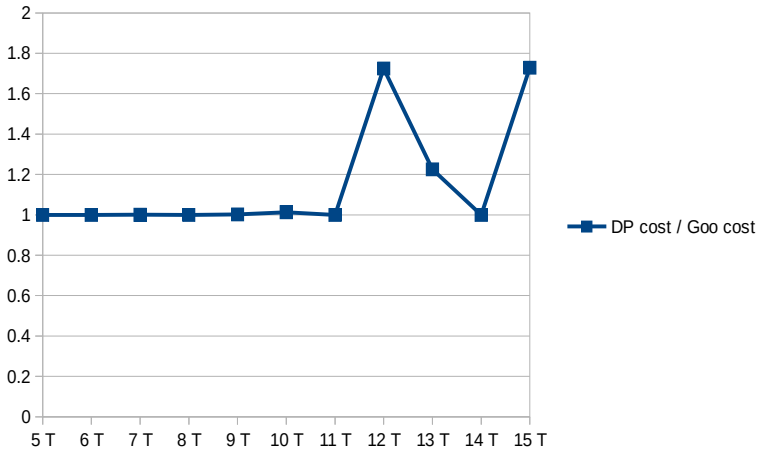

Fig. 7: Goo inaccuracy for naive random queries
Source: Own creation

Since two sample points might not be enough to proof this theory we did the same thing for 100 queries each of sizes 5 to 15, we generated naively by setting each predicate and cardinality to a random number, the predicates being in range]0;1] of course. The results are shown in Fig. 7: Goo inaccuracy for naive random queries and again we can detect a loss in quality for larger queries, although the table sizes 13 and 14 act out, which might be related to the fact that if GOO finds the right solution depends on semantics of the query and not only it´s size. Exactly that can be seen if we look at the two diagrams, GOO on average creates way worse solutions for our auto generated queries compared to the ones that follow a more realistic approach.

2.3.2 Optimization Time

But now we get to the trade-off GOO gains for delivering worse quality plans: execution time.

Fig. 8: Runtime with realistic queries
Source: Own creation

In Fig. 8: Runtime with realistic queries we tested the small sample of realistic queries for optimization time, the samples with 5 tables and 15 tables, here we can already see the big gain in optimization time for the DPsize executions, coming with the increase of query size, yet the two sample points are not enough to come to any conclusions. Since the realistic queries need way less time to execute than the ones we generated, we will continue the testing with only the ones we randomly generated.

Fig. 9: GOO vs. DP runtime in seconds shows the optimization time of GOO and DP in seconds when executing our generated queries of sizes 5 to 14, we left 15 out because with an average of 312 seconds per query our diagram would become less readable. Looking at the graphic it becomes quite obvious that DP really suffers from it´s exponential nature in this benchmark. While the DP runtime increases exponentially the GOO seems to barely change it´s runtime in Fig. 9: GOO vs. DP runtime in seconds. If we take a close look on GOO in Fig. 10: GOO runtime in milliseconds which depicts the same values only for the GOO execution in milliseconds, for this graph it is not completely obvious since we are working with very small values, it could show anything between linear and cubic complexity but it is still detectable to be better than an exponential graph.

The DPsize are meeting our exceptions of exponential growth, and while we need to test the GOO implementation for a greater range of values to determine it´s growth function it is still very clear that GOO saves a lot of optimization time compared to the DPsize implementation.

It has to be noted though, that since we measured the time once before and once after an execution of the different algorithms we must not interpret a lot into the exact values of the small execution times due to inaccuracies, but the trend we determined should still be valid. All our findings should in theory also apply to a fully extended ERM approach only that while its quality of plans will never be worse than GOO it can produce better ones. Runtime wise it will be slower than GOO but the speed loss is expected to be merely significant since GOO and the ERM extension have the same complexity as explained before.

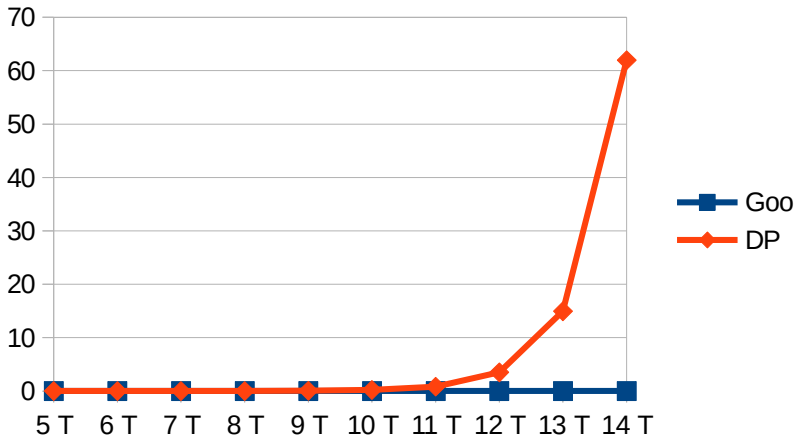

Fig. 9: GOO vs. DP runtime in seconds
Source: Own creation

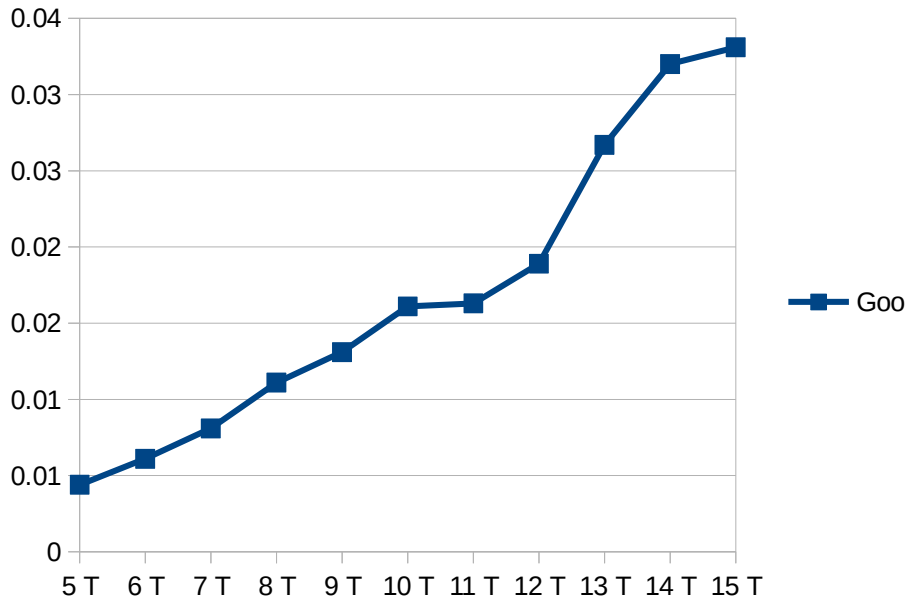

Fig. 10: GOO runtime in milliseconds
Source: Own creation

3 Conclusion And Future Work

Summing everything up the ERM heuristic, which has big similarities to the earlier greedy approach GOO, delivers good results when optimizing queries, that get worse the bigger or complex the queries get, but on the plus side is still able to optimize larger queries in feasible time, which might not be possible for exponential approaches, depending on the time constraint we have. So the ERM Approach is a good tool if we need things to get done in a certain time but can´t compete with optimal approaches, if the time we have to find our solution is not limited.

For future work we can implement the additional extension of ERM for further testing and analyse how big of an impact the push and pull extensions have on the quality of plans. Furthermore the ERM heuristic brings extensions for more complex queries to the table and can be extended to support group-by clauses, semi-join operators anti-joins and outer-joins.

4 References

1 Leonidas Fegaras. “A new heuristic for optimizing large queries”.In Gerald Quirchmayr, Erich Schweighofer, and Trevor J. M. Bench-Capon, editors,Database andExpert Systems Applications, 9th International Conference, DEXA ’98, Vienna, Austria, August24-28, 1998, Proceedings, volume 1460 ofLecture Notes in Computer Science, pages 726–735.Springer, 1998.

2 Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. ”Polynomial heuristics for query optimization.”In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum,Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra,Umeshwar Dayal, and Vassilis J. Tsotras, editors,Proceedings of the 26th InternationalConference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA,pages 589–600. IEEE Computer Society, 2010.

3 Guido Moerkotte, Thomas Neumann “Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products” pages 930-941 VLBD, September 2006,

4 Viktor Leis, Andrey Gubichev, Thomas Neumann. “How good are query optimizers, really?” In Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper and Thomas Neumann, Leis, pages 204-215, VLBD 2015

5 Neumann, Thomas, and Bernhard Radke. "Adaptive optimization of very large join queries." Proceedings of the 2018 International Conference on Management of Data. ACM, 2018.