Final Report 2ID35
Database Technology

C. Lambrechts - 0733885 - c.lambrechts@student.tue.nl
K. Triantos - 0852612 - k.triantos@student.tue.nl
J. Wulms - 0747580 - j.j.h.m.wulms@student.tue.nl

June 22, 2014

Abstract

This report contains the final deliverable of the project for the course 2ID35 Database Technology. We provide some background for the paper we are studying, work out the research problems that have been addressed and what results have been claimed by the paper. We then proceed by giving an overview of what we have done in order to verify their results. We end with a discussion of the results.

1 Introduction

Optimizing a query can be done in several ways. The operation that is the computational most expensive is the join operation. So it makes sense to try to make this operation faster. Changing the order in which joins are performed is a common approach. For this approach cost approximations are computed before actually performing the joins. This approach requires the development of a cost model, an assignment of an estimated cost to each query processing plan and searching in the (huge) search space for the cheapest cost plan. There are also approaches that do not use cost plans. One of the approaches focusses on minimizing the number of joins instead of using a optimal join order. This approach requires a homomorphism test, which is NP-complete. An other approach focusses on the structural properties of the queries. It will try to find a project-join order that will minimize the size of the intermediate results during query evaluation. Practical it means, use only the data that you need and remove the rest as soon as possible.

The paper [1] we are reviewing uses such a structural approach. The authors of this paper try to push down the projections, such that the attributes that are not needed are projected out as early as possible. The projection pushing strategy has been applied to solve constraint satisfaction problems in Artificial Intelligence with good experimental results. The input to a constraint-satisfaction problem consists of a set of variables, a set of possible values for the variables, and a set of constraints between the variables; the question is to determine whether there is an assignment of values to the variables that satisfies the given constraints. In the database area, this means that you are searching for entries that satisfy the constraints on their attributes.

Optimizing the query manually, often focusses on reducing the search space before the join operation. In this fashion the number of entries used in the join is less, which can result

in a huge performance gain. Most of the time the projections are pushed down by selecting as soon as possible or pushing the projection to a sub query. It is possible to make a sub-query that projects out all irrelevant information and tries to reduce the intermediate results. Optimizing the query in an automated and structural way looks promising and practical.

We start by presenting the problem description found in the paper in Section 2. Then the results claimed in the paper are discussed, Section 3. To verify the results we present our methodology in Section 4. Section 5 presents the results of our experiments and Section 6 discusses them. Section 6 also discusses the claimed results of the paper [1] in comparison with our results. Section 7 concludes this report and presents our conclusion.

2 Problem Description

In this section we describe the problem that is addressed in the paper, but we also state the solution that the authors proposed. We choose to do this, since the solution in the paper is actually part of our own validation problem.

As mentioned in the "Project Pushing Revisited" paper [1], *join* operation is one of the most fundamental and most expensive operations in a database query. The reason is the fact that this kind of operation combines and uses tuples from multiple relations. In the research of the paper [1] an attempt is made, which focuses on structural query properties, to find the *project - join* order, which will utilize the size of intermediate results during query evaluation, in the terms of minimization.

In general, almost every database query can be expressed as a *select - project join* query, which combine *joins* with *selections* and *projections*. The main idea of the research is to choose a *project - join* order, which will establish a linear bound on the size of intermediate results. More specifically, it is proven experimentally by the authors that a standard SQL planner spends an exponential amount of time on generating plans for such queries, with rather dismal results in terms of performance and without taking advantage of projection pushing. So the main focus of whole project is to study the scalability of various optimizing methods and to compare the performance of different optimization techniques when the size of the queries is increased.

Below we give a description of the techniques, which were tested during the papers [1] research, in order to clarify what will be verified in our own research. The following techniques must be implemented for verification. More specifically, the techniques, which will be tested, are the following:

- Naïve approach
- Straightforward approach
- Projection Pushing and Join Reordering
- Bucket Elimination

These methods will be tested on queries that solve the 3-COLOUR graph problem. The goal is to find a label out of a set of three different labels, such that each pair of vertices

that have an edge between them, do not have the same label. This is realised by creating a table for each possible edge, which has a record for each possible valid colouring for these two vertices. This comes down to a table of 6 records, which are all possible combinations of 2 different colours out of the 3 possible colours. If we now try to join all the tables for which there are edges in a graph, we solve the 3-COLOUR problem. When the join results in a table without records, there is no possible colouring. When there are records, the values in the records give a correct colouring/labelling. Furthermore, there are multiple types of graphs considered: fully random graphs, augmented path graphs, ladder graphs, graphs combining ladder and augmented properties, and augmented circular ladder graphs.

The Naïve approach constructs a query where all the tables are listed in the FROM section and in the WHERE section conditions are stated that show which attributes should be equal. The Straightforward approach suggests a specific query structure, which takes advantage of join operation, in order to minimize compile time. In the paper is stated that naïve queries are exceedingly difficult to compile and compile time is four orders of magnitude bigger than execution time. In order to get around this ineffectiveness, researchers propose to explicitly list the *joins* in the FROM section of the query, instead of using equalities in the WHERE section as in the naive approach. In this case, the order in which the relations are listed then becomes the order that the database engine evaluates the query. This technique effectively limits what SQL Planner can do and therefore drastically decreases compile time. However, the straightforward approach still does not take advantage of projection pushing. Consequently, it was found that query execution time for the naive and straightforward approaches are essentially identical; the chosen join order is apparently no better than the straightforward order.

Projection Pushing is a step further than the straightforward approach. The idea is to produce an early projection, which would reduce the size of intermediate results by reducing their arity, making further joins less expensive, and as a result reducing execution time of the query. Early projection in SQL can be implemented with the use of sub-queries. This method processes the relations of the query in a linear fashion. However we can advocate that this technique can be optimized further. Since the goal of early projection is to project variables as soon as possible, reordering the relations may enable us to project early more aggressively. So it would be more effective if, during each step, there is a search for an atom that would result in the maximum number of variables to be projected early. According to Join Reordering, a permutation could be chosen so as to minimize the number of live variables into intermediate relations. Once this permutation is computed, the same SQL query can be constructed as before, but this time with permutation order.

A more heuristical technique regarding reduction of the arity of intermediate relations is the Bucket Elimination method. This method represents the query as a *join*-expression tree which has set of attributes as nodes, in order to describe an evaluation order to the join operation. More specifically, *join* operations are evaluated from the lower to the higher one and projection operations are applied for this specific evaluation as soon as possible. To clarify how this method works let's take an order of n query attributes. For these we build n buckets, one per attribute. Now consider these buckets iteratively, from the last to the first one and eliminate the latest bucket every time (bottom up method). During these iterations there are several relations inside the considered bucket. There is a specific attribute, iconic for

this bucket, in all these relations. For all these relations, their join is computed and attributes are projected out. The attributes that are projected out do not belong to the target schema, which we want in the end. The result can be empty, and in that case the result of the whole query is empty. If the result is not empty, we place the resulting relation in the highest bucket, which has an attribute in the joined relation that is iconic for the bucket. If this process terminates with there still being a relation in the last bucket, we have found the non-empty result of the query.

3 Claimed Results

In order to verify the paper we are looking into, we need a precise overview of the results that are claimed by the authors of the paper. This way we are able to make clear comparisons between the claimed results and our own.

3.1 Paper results

We first look at the results for the 3-COLOUR graphs where order is fixed and the density scales:

1. The curves for Boolean and non-Boolean queries have roughly the same shape.

2. At first the running time increases as density increases, because of an increased number of joins.

3. Eventually the size of intermediate results becomes small or empty, and additional joins have little effect on overall running time.

4. At low density each optimization method improves upon the previous. For denser instances, optimizations using early projection lose their effectiveness.

5. Bucket elimination completely dominates the greedy methods.

Proceeding with the results for 3-COLOUR graphs where density is fixed and the order scales. The density is fixed at 2 values, and the authors assume that the lower value is most likely associated with 3-colourable graphs, and the higher density with non-3-colourable graphs:

6. All methods show exponential increase in running, when order is increased. (This is shown by a linear slope in log-scale.)

7. Bucket elimination maintains a lower slope in log-scale, in comparison to the other optimizations. This lower slope in log-scale translates to a strictly smaller exponent, so we have an exponential improvement.

The next focus of the report was order-scaling experiments with structured queries. We first look into augmented path instances:

8. Bucket elimination is again the best, but early projection is competitive for these instances, because the problem has a natural order that works well for early projection.

9. For non-Boolean graphs the optimizations do not scale as well as for Boolean graphs. This is due to the fact that there are $\geq 20\%$ less vertices to exploit in the optimization. Early projection and bucket elimination still dominate the other optimizations in this case.

The final claims are about the results for ladder graph instances, and augmented ladder instances:

10. For ladder instances, the heuristic for reordering is not only unable to find a better order, but actually finds a worse one.
11. Furthermore, ladder instances give results very similar to augmented path instances.
12. Augmented ladder instances shows even more differences between optimization methods.
13. Non-Boolean cases for augmented ladder instances struggle to reach order 20 with the faster optimizations.

The conclusion for all these results is that bucket elimination dominates the field at every turn with an exponential improvement. In a discussion about future research areas, the authors also claim that they found results consistent with 3-COLOUR queries, when using 3-SAT and 2-SAT to construct queries.

3.2 Verification

In our verification, the main claims we want to verify are the domination of bucket elimination and the exponential improvement it shows in the paper. The next step is going into the details of all the different query types (random and structured), and getting consistent results there, or finding out why we have different results. We will not look further look into queries constructed from other sources than 3-COLOUR, such as 3-SAT or 2-SAT.

4 Methodology

This section will elaborate on the steps we took to verify the results in the paper. The following steps were taken:

- Generate graphs similar to the ones used in the paper.
- Implement the proposed algorithms to create optimized SQL queries from the graphs.
- Send query to SQL engine and measure the execution time

Figure 1 gives an overview of the process and how the results are passed. The generation and translation of graphs are implemented in Java. The idea is to have a graph generator, which outputs the graph we want to solve. Specifically, we pass a list of graphs, which will be the graphs used for a single experiment, varying in graph order or density.

The graph translation part takes the graphs as input and generates SQL queries for the graphs, according to the algorithms proposed in the paper. This part thus returns several SQL queries, in a text-file and can be passed directly to the next part in the process. This

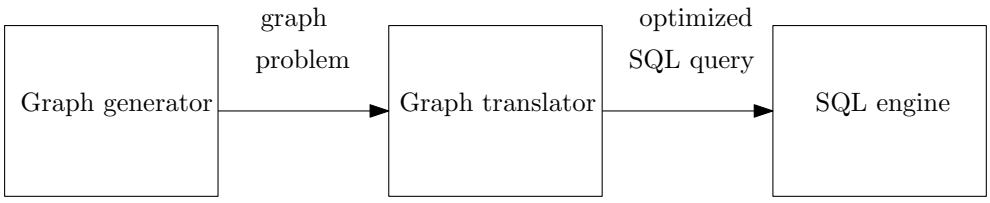

Figure 1: All steps in the process of verification

is the biggest part of the assignment, so we want worked in parallel on the different algorithms.

The queries are passed to a SQL engine. We have chosen to use PostgreSQL, to stay close to the tools that were used in the paper. However, we no longer have access to the older version of Postgres that is used by the authors of the paper, so we choose to use a newer version, namely 9.3.4. The newer version can be more optimized, so we have to take this into account when comparing results. Furthermore, we know that the authors used a Linux cluster of Itanium II, processors with 4GB of memory. We have no access to the cluster and no further details about the hardware are available, so we have to use our own machines. The machine we will use has 32GB of RAM, but we are unsure whether our set-up, using an Intel Core i5-4670K (3.40 GHz), can match the computational power of the cluster.

4.1 Bucket Elimination

Bucket Elimination method represents the query as a join-expression tree which has set of attributes as nodes, in order to describe an evaluation order to the join operation. More specifically, join operations are evaluated from the lower to the higher one and projection operations are applied for this specific evaluation as soon as possible. The name of this technique is due to the fact that it creates a bucket per variable, which investigates. To clarify how this method works lets take an order of n query attributes, for which we build n buckets, one per attribute. Inside $bucket_i$ we put all the functions which mention attribute or node i. The follow algorithm describes the general idea of the technique.

1. Process buckets backwards according the given order of nodes.
2. Eliminate the node in the bucket from subsequent computation.
3. Place the computed function from the bucket into the highest variable or node in its scope bucket.
   Given order: A, C, B, F, D, G
   Backwards order: G, D, F, B, C, A
4. Eliminate ONE bucket per time (from the n^{th} bucket to the 1^{st}.

Preparation for Bucket Elimination Algorithm execution In order to execute the algorithm we have to define an order of nodes as it is mentioned in step 1. Thus we find the MCS order of the vertices. MCS is the acronym of Maximum Cardinality Search and MCS algorithm chooses iteratively vertices. The way in which this algorithm produce the output order is explained as the next (unnumbered) vertex to be chosen: the vertex with the most already numbered neighbours. The follow algorithm describes how the algorithm works.

Bucket Elimination Algorithm Execution Given the MCS order of the vertices we create a bucket for each vertex and now we consider these buckets iteratively, from the last to the first and eliminate the latest bucket every time (bottom up method). The Bucket of the vertex v_i stores:

- Every edge, which contains vertex v_i. $edge(v_i, X) edge(X, v_i)$
- Every query regarding vertex v_i

During these iterations, inside the considering bucket there are several relations, where there is a specific attribute, iconic for this bucket, in all these relations. For all these relations, their join is computed and attributes are projected out, if they do not belong to the target schema, which we want in the end. For this reason, each bucket has live vertices. These are:

- Every vertex, which appear inside the edges of the bucket. $edge(v_i, LiveVertex) edge(LiveVertex, v_i)$
- Every vertex, which appear inside SELECT section of buckets queries

So, as mentioned above, we process these buckets in descending order according to MCS order in the following way. A bucket is processed by this way:

1. Create the following query
   SELECT all the *live vertices* of the bucket
   FROM all *edges* and *queries* of the bucket
   JOINing them
   ON *straightforward* approach conditions
2. Find the destination of the above query between the live vertices of the bucket. This means that we have to find the maximal vertex in querys SELECT section. With the term maximal, we indicate the vertex with the position which is closer to the first one according the MCS order.
3. Move the query, which is created during the first step, to chosen vertexs bucket and then eliminate the processed bucket.

When we reach the final bucket we will not execute the 2^{nd} and the 3^{rd} step, but we will change the SELECT section, with SELECTing only one vertex in Boolean case or SELECTing the free vertices in non-Boolean case.

The result can be empty, and in that case the result of the whole query is empty. If it is not empty, we place the resulting relation in the highest bucket, which has an attribute in the joined relation that is iconic for the bucket. If this process terminates with there still being a relation in the last bucket, we have found the non-empty result of our query.

4.2 Other approaches

Naive The naive approach put all edges in the FROM clause and puts all the conditions in the WHERE clause. This means that for each node the value must be the same. This is done by forcing this equality for the nodes on each edge only on the first edge the node occurred in.

Straightforward The straightforward approach uses the same idea, only uses the JOIN operation instead of all the edges in the FROM clause.

Early Projection and Reordering The early projection approach tries to project out all nodes that are not occurring in the later part of the computation / query. It does so by creating a sub-query of the query and uses this sub-query to continue building the query. The reordering approach just changes the order in which the nodes are projected out.

5 Results

This section presents the results from the experiments we performed. We did not perform the analysis for the different types of graphs and also did not include bucket elimination. This is due to the fact that the implementation for these are not stable. So we first start with the random graph type in Subsection 5.1. Then we will give a brief explanation why the implementation is unstable for the otehr types of graphs in Subsection 5.2 and for bucket elimination in Subsection 5.3.

5.1 Random Graphs

This subsection presents the results for the random graphs. In each graph the dots represent the measurement and the dotted line the trend line. The y-axis always represents the time in nanoseconds and the x-axis represents the variable quantity. So in case of the fixed order this is the density and in case of the fixed density this is the order. In each graph the different approaches are labelled with a colour. Blue stands for the naive approach, orange for straightforward, grey for early projection and yellow for the reordering approach.

Figure 2 shows the results for the experiment in which the order was fixed. The fixed value for the order is 20. The density ranges from 0.5 to 12.5 with steps of 0.5. Figure 3 presents the results for the experiment in which the density was fixed at 0.3. The order ranges from 10 till 27 with steps of 1. Figure 4 present the result when the density is fixed to 0.6. The order, again, ranges from 10 till 27 with steps of 1.

The ranges presented in the graphs are maximal for our configuration and implementation. When we try to exceed beyond this point the RAM start slowly filling up. When we look at what takes up this increase of RAM it is PostgresSQL and not Java. Therefore we know that it is not the query generator but the query executor. In Section 6 we will come back to why we think this happens.

5.2 Other Types of Graphs

As already mentioned before we were not able to run the experiments for the other types of graphs. This is due to the instability of the current state of the implementation. More specifically, the query generator has some issues. The graphs that are generated are generated correctly, but the query generators uses some attributes of the graph data structure and these appear not to be working properly.

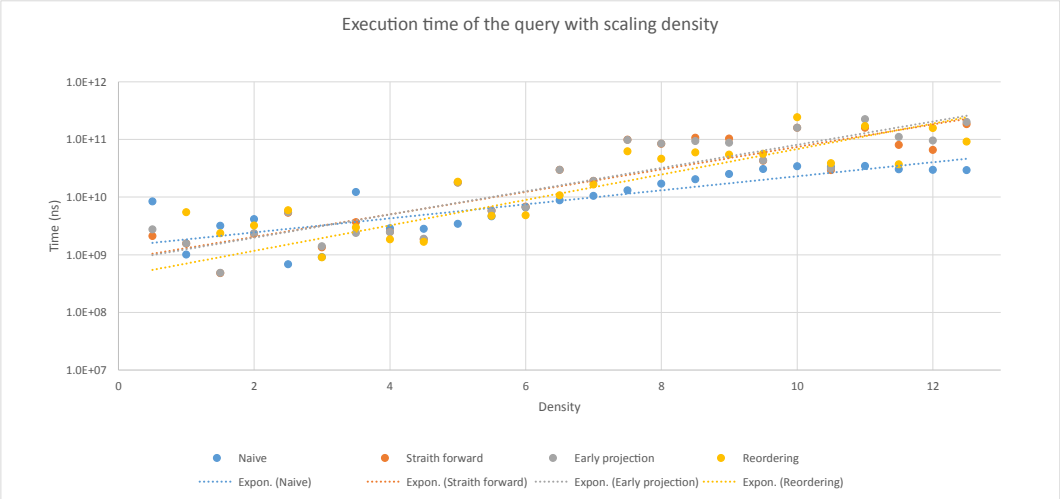

Figure 2: Graph presenting the results for the random graph type with a fixed order (20)

Figure 3: Graph presenting the results for the random graph type with a fixed density (0.3)

5.3 Bucket Elimination

Bucket elimination was also not possible to include in the experiments. We tried to focus on the bucket elimination approach as the authors of out paper [1] claim that it is better than all the other approaches used. Unfortunately we did not had enough time to fix all the bugs that popped up in the last two weeks.

6 Discussion

In this section we will discuss the results and try to verify the claims made by the authors of our paper [1]. We will start by discussing our findings and then we will compare them to the findings of our paper [1].

Figure 4: Graph presenting the results for the random graph type with a fixed density (0.6)

6.1 Our Findings

From the graphs presented in Section 5 (Figure 2, Figure 3 and Figure 4) we can see that all approaches are exponential in performance. There are differences between the approaches however. The naive approach is faster then the other approaches in the long run. It starts slower, but in the long run it becomes faster. We can see this by looking at the steepness of the trend line. This also hints us that the naive approach is faster for the fixed density of 0.6, but the steepness of the trend line is higher. So in the long run it will become slower. The other approaches seem similar in performance. Those trend lines are almost on top of each other.

We think this has to do with the combination of the density and the order. A higher density means that it is less likely to be 3-colour able. With a low density the naive approach benefits from the fact that there are a lot of nodes not connected to each other. Unconnected nodes to not lay constraints on each other. This means that when a unconnected node is added, the Cartesian product needs to be computed. Due to the way the naive approach works, the Cartesian product is already created and only needs to be filtered on. This could also explain why the other approaches are slower. The other approaches uses joins, which are not optimal to compute Cartesian products. They are optimized for joining tables. Which is beneficial in the higher density setting. More connected nodes, means more joins.

Besides these explanation, we think there could be another explanation for the differences. We generate the queries and send them to PostgreSQL, which executes them. It is almost certain that PostgreSQL tries to optimize the queries, before he actually executes them. The naive approach is quite clear and simple and therefore it is quit likely that the optimizer finds a way to make it faster. It is likely that the optimizer notices that he can rewrite the query, with some smart order of joining for example. For the straightforward approach this could be true, but it already uses joins. The order of the joins is already specified, which would make it harder for the optimizer to optimize. For the early projection and reordering approach it is unlikely that the optimizer finds a way to optimize it. Those queries are complicated and are already fixed by the query generator.

6.2 Comparison of the Findings

Before we start the comparison, lets filer out all comparison that are actually applicable to our experiments. The conclusion of the authors of our paper [1] was that bucket elimination dominates. We cannot check this conclusion due a unstable implementation, as mentioned before. This means that we can only verify or reject a claim for the random graph type and the naive, straightforward, early projection and reordering approach. The claims we can verify can be found below:

2 At first the running time increases as density increases, because of an increased number of joins.

3 Eventually the size of intermediate results becomes small or empty, and additional joins have little effect on overall running time.

4 At low density each optimization method improves upon the previous. For denser instances, optimizations using early projection lose their effectiveness.

6 All methods show exponential increase in running, when order is increased. (This is shown by a linear slope in log-scale.)

The general claims about the increase in performance when density increases (2 and the exponential increase in performance when order increases 6 are verified. This can be seen in all graphs (Figure 2, Figure 3 and Figure 4) as all trend lines are increasing.

The next claim we can verify, is the claim about the low density (4). We can say that we cannot verify this claim, as each optimization performs less than the naive approach. With the results we could even say that we reject the claim. As mentioned before this could be caused by the optimizer of PostgreSQL and by the number of joins that compute a Cartesian product.

The claim about the intermediate results (3) is harder to verify. We can implicitly verify it. We noticed that the queries that uses joins are performing worse than those that do not use joins (the naive approach). We know that joins are better at performing joins. There are two paths we can take to verify this claim. The first is by using the variables that are used in the join and the second is by using the intermediate sets. When the variables that are used for the join are not common, the Cartesian product is computed. If the intermediate results would be smaller or empty, this operation is more efficient. Hence the performance would be better. This is not the case as it performs worse than the naive approach, even in the long run. Now lets take the other path. If the intermediate results becomes smaller or empty, there less common variables to join on. Which would imply that there are more Cartesian products computed and thereby the performance becomes worse. Which can be seen in the results. Both the paths argue that this claim can be rejected.

7 Conclusion

Our experiments show that there are some differences between the results. These differences do not verify some claims from the paper [1] we reviewed. We should note that we were not able do run all experiments, more specific we did not experiment bucket elimination and the special types of graphs. We think that the differences can be explained by the PostgreSQL

optimizer, which is capable of optimizing the more simple and naive queries and cannot optimize the complex queries. Another reason why our results differ has to do with the computation of the Cartesian product. Using JOIN operations to compute this is a lot slower than with the FROM and WHERE clauses. Overall we are able to verify that all approaches are exponential in running time.

References

[1] B. J. Mcmahan, G. Pan, P. Porter, and M. Y. Vardi. Projection pushing revisited. In *In Proc of EDBT04*, pages 441–458, 2004.