Finding and materializing common subexpressions among queries in a query workload

Mark Pasterkamp

Finding and materializing common subexpressions among queries in a query workload

by

Mark Pasterkamp

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

to be defended publicly on Friday February 7, 2020 at 3:00 PM.

Student number:	4281500	
Chair:	Prof. dr. ir. G. J. P. M. Houben,	TU Delft, WIS research group
Academic supervisor:	Dr. A. Katsifodimos,	TU Delft, WIS research group
Committee member:	Dr. C. Lofi,	TU Delft, WIS research group
Committee member:	Dr. J. S. Rellermeyer,	TU Delft, DS research group

TUDelft

Declaration of Authorship

I, Mark Pasterkamp, declare that this thesis titled, “Finding and materializing common subexpressions among queries in a query workload” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ***Mark Pasterkamp***

Date: ***31 - 01 - 2020***

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Finding and materializing common subexpressions among queries in a query workload

by Mark Pasterkamp

Most queries in a collection of queries, also called a query workload, to some degree have parts of their intermediate execution steps in common. These intermediate execution steps, also called subexpressions, provide the opportunity to further optimize query workload execution in addition to the already existing query optimization done by the DBMS. A lot of research has been done into this topic however most of that research is either proprietary or just not applicable to big query workloads. In this thesis I developed a simple yet effictive heuristic algorithm to quickly find common subexpressions and materialize them to disk using open source software with results showing a significant increase in performance.

Acknowledgements

First of all I would extend my sincere gratitude towards my friends. Not only were you always there for me during these hard and stressful times, but also way before back when I first started university. Thanks to you, I have become the person I am today of which I will always be grateful. The experiences and emotions we shared as well as the hardships we faced during university have bonded us. Now with my graduation finally in sight it is time for this chapter of my life to end and to start a new journey.

I am also very grateful for the support from my family. They were always there for me when I needed them and their unconditional support has helped me tremendously throughout the thesis. I would also like to thank my brother, Stephen, for staying with me after my parents decided to move out. Living on your own is a challenge on its own and I am glad we had each other to figure things out.

Finally I would like to thank my supervisor, Asterios Katsifodimos, for helping and guiding me during the process. It is easy to let lose yourself in your thesis and Asterios helped me putting things into perspective. I distinctly remember a time were I was extremely worried about missing deadlines. During a meeting he then told me that I should not need to worry that much about these things, they can happen. And unlike medical practitioners where failing objectives could have large consequences, failing a deadline is just unfortunate but can happen from time to time. Although it is never the intention to miss a deadline, it did help me to not focus too much on the consequences and just keep going.

Additionally I would like to thank you, the reader, for taking the time to read through my thesis. I hope this thesis will provide you the answers which you seek.

Sometimes people wonder what their younger self would say about who you are today. I would like to think my younger self would be proud of who i am today, which would not have been possible without all your support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An important factor in data warehousing is performance. Especially with the ever growing amounts of data (Gantz and Reinsel, 2012), it is important that data is parsed quickly to accommodate larger volumes of data. One way to parse information faster is to have a better machine. This is called vertical scaling. This used to be an effective strategy with the industry following Moore's law (every 2 years, the amount of transistors on a chip doubles). However, these last few decades we can see a slowdown in Moore's law (Simonite, 2016). It has become clear that following Moore's law is becoming infeasible from an economics standpoint (Mack, 2011).

What has been done to increase performance instead is joining multiple computing systems together in a network. This is called horizontal scaling. These so called distributed systems can be relatively cheap to setup compared to a single system with similar performance by using commodity hardware. A big challenge of these systems is how to keep them operating with each other in an unreliable network (Tanenbaum and Steen, 2013). To keep these distributed systems running during unreliable networking one has to sacrifice other parts of the system.

1.0.1 Eliminating subexpression

As can be seen vertical scaling also has its drawbacks and by continuing expanding on its size one gets a lot of overhead in communication between its nodes. However, there are other alternatives to increase the query performance. Another impact on the performance of data warehousing is the amount of duplicate work. Perhaps due to lack in coordination or perhaps due to other reasons, it can occur that different clusters in the network can the same jobs multiple times. For instance Microsoft noted that 45% of the daily jobs submitted by 65% of the users contained commonalities (Jindal et al., 2018).

For this thesis we are going to tackle the problem of finding and eliminating these duplicate computations. Focusing on database queries, we will transform these queries into their respective tree of relational operators. Then, from that tree extract the subexpressions and find materialize those subexpressions with most value with regard to the memory constraint. Finally, using Apache Calcite, rewrite the queries to use these materialized subexpressions.

For this task it is important to keep in mind that time spent looking for a subset of subexpression to materialize could also be time invested in executing the queries. This trade off between finding better subsets of subexpressions and time spent finding them means that finding a good subset of subexpressions fast is more important than finding the optimal subset of subexpressions.

1.1 Subexpression selection problem in literature

Various research has been done into the subexpression selection problem. The problem with most research done however is that they do not translate well into a production solution. Either the developed solution is proprietary (Jindal et al., 2018) or just not practical due to the performance limitations (Finkelstein, 1982).

What can be seen by investigating this problem in the existing literature is that the taken approach is generally the same. Subexpression are extracted from the various queries and assigned each with a value and weight based on the size and the difference between scanning and computing respectively. The difference between the approach taken by the different studies into this problem can be found in the implementation in the different approaches.

1.2 Problem statement

As the data keeps accumulating more and more (Gantz and Reinsel, 2012) and gaining insights into said data keeps getting harder and harder, smarter approaches need to be taken to ensure reasonable performance. One of such approaches is by leveraging the fact that a lot of queries in a query workload can have similar elements. As mentioned before, with large query workloads there is a high chance that some queries might share some computations with each other. Similar to dynamic programming (Kleinberg and Tardos, 2013) we try to isolate recurrent subproblems, overlap in query logical operator trees, and try to precompute and store those partial results.
For instance, say there are two queries: q_1 and q_2, with q_1 defined as

find all actors involved in action movies

and q_2 defined as

find all actors involved in European action movies

It can easily be seen that q_1 and q_2 have similarities in their queries, since both queries would have to access all actors involved with action movies (which is further refined in q_2)

In this thesis we are going to look at the problem of finding similarities between queries in a query workload and how to evaluate the potential gain of materializing those similarities to disk. Furthermore, we also want to look at how we can leverage these materialized similarities between queries by rewriting the existing subexpressions.

Given a query workload Q and a memory constraint M. What we want to try to solve is the maximization problem of finding set of common subexpressions $S = \{s_1, s_2, \ldots, s_n\}$ such that $|S|$, the memory size of the set of subexpressions, is at most equal to M while maximizing a certain objective function (total profit of materializing S) using open source systems.

Finally, given a set of materialized subexpressions S how can we rewrite the existing queries to use the materialized subexpressions in S.

1.3 Main research questions

As previously mentioned, even though this problem has been studied a lot in previous literature, solutions to this problem have not been widely adopted. Mostly because it is either not fast or proprietary. To overcome these shortcomings, the software used in this thesis is entirely open source. This is why this thesis is also partly a feasibility study, making the first and main research question:

RQ1: ***How can we implement the existing state of the art of finding and materialing common subexpressions among queries in an query workload in 2019 using open source software?***

In order to find an answer for this research questions other problems need to be solved as well. First we need to find candidate subexpressions, subexpressions suitable for materialization, from the query workload. This leads to the second research question of the thesis:

RQ2: ***How can we efficiently extract candidate subexpressions from a query workload?***

Finally, after extracting the set of candidate subexpressions there needs to be a way to evaluate which subexpressions to materialize with regard to memory constraint while also not wasting too much time. Preferably, this process happens fast so to not waste time which could have been used to execute the different queries. Thus, the final research question will be:

RQ3: ***How can we efficiently evaluate candidate subexpressions to find a good solution?***

With a good solution we refer to finding a subset of subexpressions to materialize, ideally providing a speedup in query execution time. For the third research question I opted that finding a good solution quickly is much more important than finding the optimal solution in considerably longer time.

RQ4: ***How can we rewrite queries to use existing materialized views?***

Creating a query rewriter is no easy task, which is why I chose to use an off the shelf one instead. Apache Calcite have already implemented an query rewriter based on based on Goldstein and Larson, 2001

1.4 Main contributions

While similar work has been done by others, they either focus on smaller query sizes or are closed source solutions. This research has been conducted with an open solution in mind using an open source query engine rewriter (Apache Calcite).

The view selection problem is an NP-hard problem (Agrawal, Chu, and Narasayya, 2006). Properties of algorithms to solve problems are that they are exactness, general and relatively fast. For these NP-hard problems however, we cannot have all 3 Properties at the same time. we will need to sacrifice one property. Having a relatively fast algorithm is most important since every second spent finding a solution is time and resources which could have been spend computing queries. Also, to ensure a relatively simple application to current domain problems, a general algorithm has also been prioritized in favor of exactness.

Concludingly, my contribution to the current state of the art is a general fast solution for finding and materializing candidate query subexpressions using open source solutions.

1.5 Thesis structure

The remaining part of the thesis is structured as follows. First, in chapter 2 we will dive deeper into the existing literature and see what has been done. Then in chapter 3 we will take a look at the general approach and methodology used followed by the implementation details in chapter 4. In chapter 5 we are going to look at the evaluation of the results obtained from the experiments. Then in chapter 6 we will discuss the threats to the validity of this research. Chapter 7 will discuss the potential future work with regard to the research, and the research goal in general. Finally, in chapter 8 we will conclude the thesis.

Chapter 2

Related work

The objective of this work is to find a set of common subexpressions and materialize them to disk to avoid having to recompute them. This problem, the problem of finding an efficient execution plan for multiple queries, is called Multi-Query optimization. Traditionally, databases were more focused on finding local query optimization. As the name implies, the Multi-Query optimization problem describes the problem of finding a more global optimization for a group of queries (Sellis, 1988).

Multiple approaches for this problem can be found in literature. One of such approaches is by analyzing the query access path to find common subexpressions to materialize (Sellis, 1988, Jarke, 1985, Mistry et al., 2001).

In this chapter we are going to look at the established literature into this problem. The problem what we are trying to solve, finding and materializing commonly accessed data can be divided into two different subproblems: the view selection problem and the subexpression selection problem. Both of these problems will be discussed as well as alternative methods for multi-query optimization like distributed public/subscribe systems and the shared workload optimization. First in 2.1 we will discuss the view selection problem followed by the subexpression selection problem in 2.2. Then in 2.3 we will look at the shared workload optimization problem. Finally, in 2.4 we take a look at the literature for distributed public/subscribe system.

2.1 View selection problem

The view selection problem is to choose a set of views to materialize such that the cost of evaluating a set of queries is minimized and such that the memory cost of these views does not exceed a specified memory constraint (Chirkova, Halevy, and Suciu, 2002). In this section we are going to discuss the existing literature with regard to the view selection problem. One group of literature tackles the view selection problem by analyzing the AND/OR-graph, so we will first discuss the AND/OR-graph and its literature. After that we will look at another approach, which is representing the data as a data cube using lattices.

2.1.1 Analyzing the AND/OR-Graph

The view selection problem describes the problem of finding slices of the data most beneficial to materialize. The view selection problem is a NP-Hard problem (Agrawal, Chu, and Narasayya, 2006). Therefore, most research into this topic has focused been focused on finding an efficient approximation for this problem.

Early techniques into this problem in data warehouses have been Focusing on analyzing the AND/OR-graph of a query (Gupta and Mumick, 2005). An AND/OR-graph of a query is an alternative way of representing the query as a direct acyclic graph (DAG). The nodes in these graphs represent the intermediate computations for a query and the edges a dependency between them. In these DAGs, the source node is the final node containing the answer to the original query and the sink nodes are the base relations being queried.

As an example, say there is query q asking for all graduate students, which are also employees. In figure 2.1 we can find 2 different graphs. Graph a is called an AND-Graph of query q. Graph b is an example of an AND/OR-Graph where there are multiple ways of evaluating query q, one way is by calculating the intermediate results of graduated students directly as in a, and the other is by using an existing view called Graduate_Students. The curved lines in these images represents an AND relation (also called AND curve) meaning that all connected edges are needed to compute the result. The difference between and AND-Graph and an AND/OR-Graph is that an AND-Graph only shows 1 possible way of evaluating the different intermediate expressions whereas an AND/OR-Graph can show multiple evaluation methods. These AND/OR-Graphs can be used to represent queries by modeling their respective logical access path. The logical access path of a query is the computation involved to answer a query made against the database including a number of intermediate views (Roussopoulos, 1982).

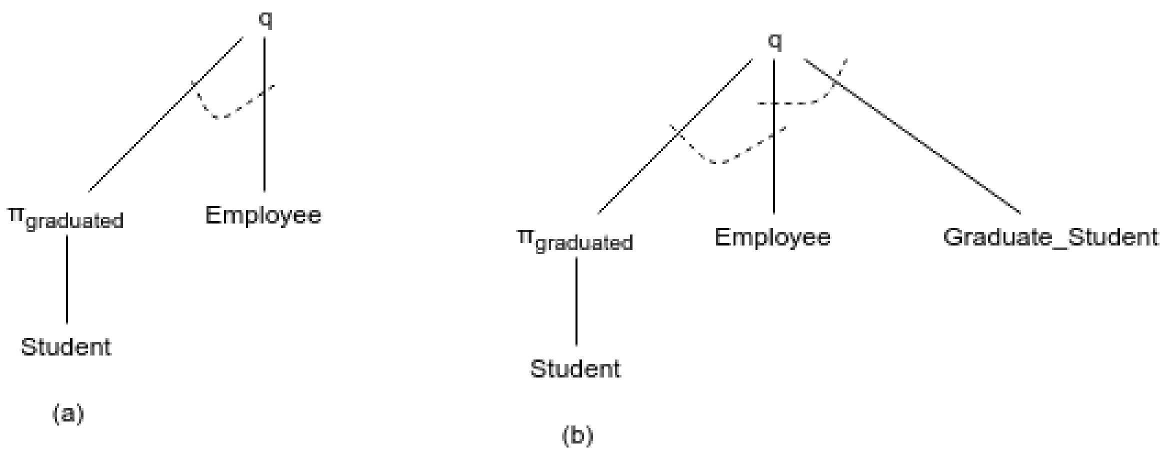

FIGURE 2.1: (a) Example of AND-Graph (b) Example of AND/OR-Graph

Obtaining the AND/OR-Graph

To create these AND/OR-Graphs, Gupta and Mumick described three general steps to transform a set of queries into an AND/OR-Graph. First, for each query $q_1, q_2, \ldots, q_n$ they create an AND/OR-Graph of all possible logical access paths, creating multiple AND/OR-Graphs $D_1, D_2, \ldots D_n$

Then, in the second step, Gupta and Mumick use these multiple graphs to construct an AND/OR View Graph. This is done by iterating over each graph $D_1, D_2, \ldots D_n$ and iteratively integrating each new graph into the AND/OR View Graph. Let G_i be the AND/OR View Graph constructed from $D_1, D_2, \ldots, D_i$. Integrating D_{i+1} involves matching each node with the AND/OR View Graph, which represent the same relational expression. For the nodes with no matching relational expression in the view graph one would need to identify whether this node can be evaluated from existing nodes in the view graph or create a new sink node in the view graph.

Finally, in the last step, Gupta and Mumick mention computing the view parameters, the access frequencies and update frequencies based on existing query workloads.

Obtaining the Views

After obtaining the AND/OR-Graph, Gupta and Mumick developed a greedy algorithm called "AO-Greedy". The algorithm is an iterative greedy algorithm behaving a lot like a knapsack algorithm. First, the AND/OR-Graph is converted to a bipartite graph $G = (Q \cup C, E)$, where Q is the set of queries and C the subset of the power set of views (the views found in the AND/OR-Graph). An edge $(q, v) \in E$ iff query q can be answered using view v.

Then, for each subset of views Gupta and Mumick compute the profit and the cost of the union of those views. Finally, after computing the profit and the cost of each subset of views found in the view graph, Gupta and Mumick transform the problem into a greedy knapsack.

Worst case, "AO-Greedy" is an exponential time algorithm but has a performance guarantee of 65% (Gupta and Mumick, 2005).

2.1.2 Data cubes

Another approach to the View Selection Problem has been done by using latices to represent data cubes (Ross and Srivastava, 1997). With this approach, one represent their data as a multi-dimensional data cube where each cell consists of an aggregation of interests (for instance total sales). Data cubes are often used in data warehouses to represent the data to its users. Ross and Srivastava use the TCP-D support decision benchmark (*TPC-DS*) as an example. This benchmark models a business warehouse where parts are bought from suppliers and sold to customers.

Ross and Srivastava describe three dimensions from this benchmark which they are interested in: **part**, **supplier** and **customer**. Using those 3 dimensions, Ross and Srivastava create a 3-dimensional data cube where each cell (p, s, c) stores the total sales of part p bought from supplier s and sold to customer c. Furthermore, an extra value to each dimension was added, ALL to be able to represent aggregated sales to allow for queries as: what are the total sales of part p sold to customer c. This query would then be represented in the cell (p, ALL, c).

Ross and Srivastava also describe a lattice framework which they use to identify which queries could be answered using the results of different views. For instance, let query q be the query asking for all sales aggregated by year, and view v be the view describing all sales aggregated by month. Although q and v are not exactly the same, we can answer query q using view v by summing up all the monthly sales into yearly sales. This is denoted by $q \succeq v$ in the aforementioned lattice framework.

The previous example is an example where data hierarchies are used to define the lattice relation. Looking at the data cube, some dimensions can be represented by multiple attributes. These attributes are organized in hierarchies. In the previous example, the dimension was time and the attributes were months and years. By recognizing these hierarchies one forms the basis of "drill-down" and "roll-up" operations. Outside of hierarchies forming a query-view dependence, one can also find more normal query dependencies caused by the interaction between different dimensions.

Optimizing Data-Cube Lattices

For these data cubes it is paramount that queries are executed within reasonable time. Deciding which cells of the data cube to materialize under memory constraint is a hard problem. Using the lattice framework, Ross and Srivastava developed a lattice diagram where views (or cells) are nodes and nodes a and b are connected if and only if $a \succeq b$. Each node also has an associated

In order to optimize the data-cube lattices, Ross and Srivastava described a greedy algorithm, which tries to find an optimal solution for materializing a set of view. Define $B(v, S)$ as the benefit of materializing view v relative to the already selected views S. What the greedy algorithm does is it starts by selecting the top view in the lattice diagram (the view on which all other views can be evaluated). Then iteratively for each other view it selects the view maximizing $B(v, S)$

To compute the relative benefit of view v for a set of previously selected views S, Ross and Srivastava assume a linear cost model for evaluating views relative to the amount of rows in that view. Although this might work to get some relative notion of cost, a good point of improvement would be to incorporate better cost estimation models than row count. Especially nowadays where there we have vastly superior hardware than at the time of publication this research.

2.2 Subexpression selection problem

Finding a set of profitable common subexpressions to materialize is not trivial and has been studied under the Subexpression Selection problem. The subexpression selection problem has a lot in common with the view selection problem, with the difference being that the solution space of the view selection problem is limited by the data and the solution space of the subexpression selection problem limited by the subexpressions. The solution space of the subexpression selection problem is a subset of the view selection problem.

One of the earlier research into this topic has been conducted by Finkelstein (Finkelstein, 1982). Just like Gupta and Mumick, Finkelstein looking into analyzing the AND/OR-Graph (2.1.1) for finding subexpressions. In their research, Finkelstein focuses on finding views (which they call temporaries) to rewrite the queries in a more efficient manner. Finkelstein describes an algorithm consisting of multiple steps to achieve this. First, from the existing queries they create the query AND/OR-Graph (which is called the query graph in the research). In order to speed up the algorithm, the next step is removing any views which will not prove beneficial. Since queries often contain predicates, the next step is to propagate these predicates via the edges to order nodes in the query graph in order to create more subexpressions as well as confine some predicates related to join predicates. Next, to match queries to different views they compare the predicates from the queries with the different views and match the views to the query satisfying the predicates. Finally, they go through all possible rewritings and choose the best one.

Contrary to the other research mentioned, Finkelstein did not take into account a memory constraint when researching this topic. Furthermore, for checking the predicates between queries and views they used a satisfyability algorithm which makes their matching algorithm NP-Hard. In their research they mentioned that this does not impact their algorithm much due to effective pre-selection of views. This does however mean that for a large amount of queries and views, Finkelstein approach might not be the ideal solution.

Similar research has been done by generating candidate subexpressions from the query input tables and heuristically remove unfit subexpressions to prune the search space (Zhou et al., 2007). The four heuristics Zhou et al. used to prune the candidate subexpressions are:

1. **remove simple subexpressions** simple subexpressions are expressions which can easily be computed and are therefor not interesting to materialize;
2. **avoid subexpressions with huge result sets** subexpressions with huge result sets have a high materialization and reading cost;
3. **merge subexpressions when beneficial** by merging subexpressions one reduces the amount of subexpressions to consider;
4. **avoid subexpressions contained by parent subexpression** when a parent expression consists mostly of the child expressions (here, Zhou et al. used 90% containment), materialize the parent expression in favor of the child expression.

For the remaining subexpressions, Zhou et al. assigned the cost of using these candidate subexpressions as the reading cost and a part of the initial creation cost. After pruning the subexpressions some of the remaining subexpressions might be conflicting with other subexpressions for the same queries. Conflicting subexpressions are multiple subexpressions that can be used for the rewriting for the same query and are therefore conflicting. To guarantee the best overall performance, taking into consideration conflicting subexpressions, Zhou et al. grouped conflicting subexpressions together in order to obtain a collection of independent sets of subexpressions. By having a set of independent subexpressions, one can conclude that when these subexpressions are chosen they are the most optimal overall since there are no other subexpressions conflicting for these queries. This grouping step is included to avoid having to iterate over the power set of subexpressions.

It can be seen that most research into this topic generally take the same approach and only differ in certain aspects of implementation. Some recent research into this topic however has tried to do things a bit differently. The driving point behind their intentions was that previous research was not able to handle the huge workloads needed for their purposes (Jindal et al., 2018) Instead, they developed an algorithm called BigSubs with the aim of providing good results while most importantly being able to handle datacenter scale workloads.

Jindal et al. modeled the subexpression selection problem as an integer linear programming problem. When choosing which subexpressions to materialize one has to keep in mind that for query q_i, materializing a parent subexpression s_j devalues any rewriting for the children nodes of subexpression s_j in the same operator tree. For instance, the logical operator tree of query q_i looks like figure 2.2. Materializing subexpression t_3 would completely remove the need to materialize either t_1 or t_2 for query q_i since they would not be used in any rewriting.

To translate this aspect into the subexpression selection problem Jindal et al. introduced the interaction matrix X. X is a symmetric mxm where m is the amount of subexpressions. Element $x_{ij} \in X$ is 1 iff subexpression s_i and subexpression s_j are interacting. Subexpression s_i and s_j are interacting if one is a subtree of the other.

Using y_{ij} to denote whether query q_i is using subexpression s_j for rewriting, z_j to indicate that subexpression s_j is selected for materialization, u_{ij} to represent the utility provided by subexpression s_j for query q_i, b_j indicates the weight of subexpression s_j with b_{max} being the maximum weight. Finally, x_{ij} representing whether

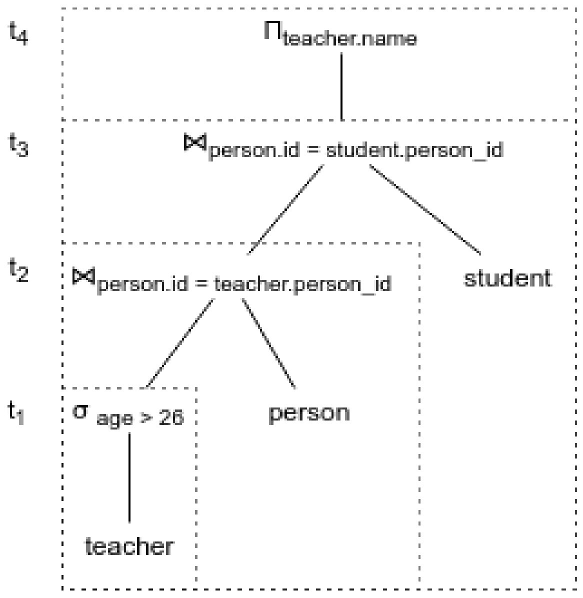
t4
Π teacher.name
t3
⋈ person.id = student.person_id
t2
⋈ person.id = teacher.person_id
student
t1
σ age > 26
person
teacher

FIGURE 2.2: The identified subexpression in the logical operator tree 3.1

subexpression s_i is interacting with subexpression x_j. With these variables, Jindal et al. transformed the subexpression selection problem into the following ILP:

$$\begin{aligned}
\text{maximize:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} u_{ij} \cdot y_{ij} \\
\text{subject to:} \quad & \sum_{j=1}^{m} b_j \cdot z_j \leq B_{max} \\
& y_{ik} + \frac{1}{m} \sum_{j=1, j\neq k}^{m} y_{ij} \cdot x_{jk} \leq 1 \qquad \forall i \in [1,n], k \in [1,m] \\
& y_{ij} \leq z_j \qquad \forall i \in [1,n], k \in [1,m]
\end{aligned}$$

The second constraint ensures that for a given query q_i, no subexpression s_j is chosen iff another interacting subexpression is already chosen.

Sadly, this ILP problem is non-linear in nature and would take quite a lot of time with huge workloads. To solve this, Jindal et al. transformed the problem into a bipartite edge labeling with graph $G = (V, E)$ where $V = Q \cup S$ the union of nodes representing the queries on the left and the nodes representing subexpressions on the right. An edge $e_{ij} \in E$ iff query q_i can be rewritten using subexpression s_j.

To ensure that the subexpression selection happens relatively quickly Jindal et al. approximated the solution in their algorithm BigSubs. BigSubs splits the problem into two different subproblems. First, using a probabilistic approach, BigSubs selects the subexpressions to materialize taking into consideration the weight of the subexpressions as well as the total utility it adds with regards to the current utility. With each iteration, the probability of a subexpression flipping (being selected or not) becomes smaller until convergence is reached.

After probabilisticically selecting which subexpressions to materialize, a smaller ILP problem is formulated to assign labels to the edges. This edge labeling solution is used to determine which subexpressions is used for rewriting. This smaller ILP problem is formulated as:

$$\begin{aligned}
\text{maximize:} \quad & \sum_{j=1}^{m} u_{ij} \cdot y_{ij} & \\
\text{subject to:} \quad & y_{ik} + \frac{1}{m} \sum_{j=1, j \neq k}^{m} y_{ij} \cdot x_{jk} \leq 1 & k \in [1, m] \\
& y_{ij} \leq z_j & k \in [1, m]
\end{aligned}$$

This process is then repeated for a specified amount of repetitions until convergence is reached (no further changes in the subexpression selection) By splitting the subexpression selection into these two separate subproblems (selecting subexpression and assign them to queries) Jindal et al. is able to create multiple smaller ILP problems allows BigSubs to parellelize the original problem a lot better making sure that approximate solutions are calculated rapidly.

2.3 Shared workload optimization

Contrary to the previously discussed research, the focus of shared workload optimization is not to find the most efficient query plan. Instead, the aim of shared workload optimization is to avoid optimizing the queries in order to maximize overlap in query execution plans. This overlap in execution plans allows for the system to combine different queries into one plan and share the results between the different queries. This method is another way to optimize collections of queries in that the overhead of calling operators like joins and tablescans only happens once for multiple queries. By keeping track of which records are needed for which queries one can aggregate the results at the and based on the queries and send the results to their respective queries. These problems are difficult to solve because not only should a decision be made on the ordering of the different operators for single queries and which algorithm to use for said operator, but also on which operators are shared between which queries.

One such research is described in Giannikis et al., 2014. Giannikis et al. developed a branch and bound dynamic programming algorithm to solve the shared workload problem. A branch and bound algorithm describes an algorithm, which can choose multiple paths based on decisions it can make and for each branch it creates it tries to decide quickly if it is a path worth pursuing based on previously found solutions or not (Kleinberg and Tardos, 2013). Since tuples are shared between different queries normal cost indicators like cpu-cost and io-cost are not generally applicable as cost functions (since a high cpu cost for an operator shared among multiple queries does not necessarily mean it is a bad decision). Instead Giannikis et al. opted to minimize tuple count as their objective function.

As mentioned previously the shared workload optimization problem is a hard problem and going through all the exponentially growing possibilities in a branch and bound algorithm would take an infeasible amount of time. Instead Giannikis et al., 2014 first created a baseline solution by using a share all approach in an ILP formulation. By saying that all operators are shared as much as possible one severely reduce the problem space making it much easier to find a baseline solution using an ILP solver. To avoid that the algorithm searches for infeasible solutions Giannikis et al., 2014 introduced 2 types of heuristics: sharing heuristics and ordering heuristics. The sharing heuristics gives the algorithm hints on which operators not to share and which can be shared across which statements. The ordering heuristics provides the algorithm with hints on how to explore the solution space with regard to ordering operators.

Finally, with these two types of heuristics and the baseline solution Giannikis et al., 2014 start to explore the solution space branching into multiple solutions. When a solution starts to have a cost greater than the baseline they close that branch and start exploring the next. When a better solution is found, the baseline gets updated until the solution space is exhausted finding the final and best solution for the algorithm.

2.4 Distributed public/subscribe systems

Other methods to improve performance for collections of queries have been Focusing on sharing the workload between different users (Karanasos, Katsifodimos, and Manolescu, 2013). These systems reduce the workload stress by sharing it among its consumers. There are nodes in the network sharing the data (publishers) and consumers consuming the data (subscribers). This can work extremely well in services like streaming services where its consumer can gain their data from the server as well as other consumers. This is advantageous in situations where an expensive operator is already applied to the data by another consumer, so one can download the data from them without applying the expensive operator oneself, resembling the likes of a peer to peer system. The goal of the system is to minimize the total resource utilization while maintaining low latency between different subscribers while respecting the given resource constrains. This is accomplished by analyzing the different requests and constructing a rewrite graph.

A rewrite graph is a directed graph $G = (V, E)$, with $V = D \cup S$ where D represents the original data publishing node (for instance, a standard database) and S the set of subscribers. An edge $(v_1, v_2) \in E$ iff the data request in v_2 can be answered using the data published in v_1. A rewrite graph can be used to represent which queries can be used to answer other queries. Figure 2.3 shows an example of a rewrite graph with 3 configurations: A, B and C. Configurations A and C show both extremes of the rewrite graph, where configuration A represent the extreme where the consumers do the least amount of work, and the original data distributor does the most (basically no distributed public/subscribe) and configuration C shows the other extreme where the original distributor does the least amount of work, and the consumers do most of the work. In configuration B one can find a more balanced approach where the distributor distributes the data to consumer 1 and 2, which in turn distribute their data to consumer 3. It is also important to take into consideration the data latency for the different subscribers. In figure 2.3 configuration C, consumer 3 has to wait until the queries of consumers 1 and 2 are finished. This latency is increased for each extra subscriber in the path between the distributor and the subscriber.

The main objective of the the distributed public/subscribe system is to find a configuration minimizing the workload on the original data distributor while also making sure the data latency remains within acceptable bounds.

The disadvantage however is that the system now has to rely on the willingness of users to share their internet with other users. Furthermore, depending on the size of the workload, the users of the system might not be able to share the workload within a reasonable timeframe.

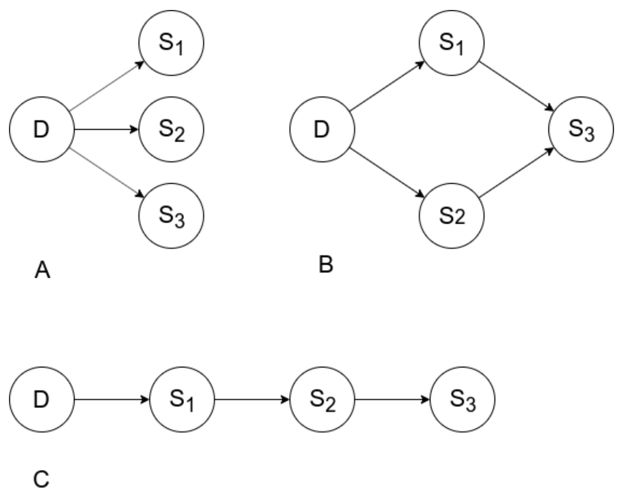

FIGURE 2.3: A: Every consumer is subscribed to the data source. B: Request of consumer 3 constructed from consumer 1 and 2. C: Every consumer subscribed to another consumer except for consumer 1.

2.5 Summary

In this chapter we have looked at the existing literature in multi-qeury optimization. First, in 2.1 we have looked at the view selection problem. Here we discussed two different approaches, one by analyzing the AND/OR-Graph in Gupta and Mumick, 2005 and another by representing your data as data cubes using lattices in Ross and Srivastava, 1997.

Then, from the view selection problem we started to look into the research of a subproblem of the view selection problem: the subexpression selection problem in 2.2. Here we saw some more research focussed on analyzing the AND/OR-Graph in Finkelstein, 1982. We also looked at heuristically pruning the given subexpressions to further reduce the solution space in Zhou et al., 2007. Most research on this topic generally comes down to rewriting the problem as a knapsack problem, but we also looked at a more recent approach where the problem is divided into multiple interger linear programming problems in Jindal et al., 2018.

After that, in section 2.3 we have looked at the completely different approach. Instead of focussing on finding the best views to precompute query results, Giannikis et al., 2014 studied the possibilities of grouping different operators together such that these operators are only executed once for each query.

Finally, in section 2.4 we have taken a look at distributed public/subscribe-systems (Karanasos, Katsifodimos, and Manolescu, 2013). These systems try to reduce the workload on the system by having its participating clients share intermediate results among each other.

Chapter 3

Methodology

In this chapter we will discuss the methodology as well as some of the terminology used throughout the thesis. First in 3.1 I will give a brief description of the definition of a query and provide a more formal explanation of what subexpressions are and how they are related to queries. Then to expand on these definitions we will look at the meaning of "query workload" in 3.2. Although already briefly discussed previously, we will take a more in-depth look at query rewriting in 3.3. Because our approach share a lot of similarities with the knapsack problem (Salkin and De Kluyver, 1975) we will look at this problem into detail in 3.4 followed by a brief introduction to the quadratic knapsack problem in 3.5. Then we will take one more look into the candidate selection problem in 3.6. Finally, we will discuss the experiment in 3.7

3.1 Queries and subexpressions

Queries are requests one sends to a database asking for specified information. For instance, the query

```
SELECT * FROM countries
```

will select all countries in the database.

Another way to look at queries is to see them as a collection of logical operators. Logical operators are operators from the relational algebra, which describe how the data is being parsed and prepared during each step of the process. Some of the more common of these operators are:

- *Select (σ)* The select operator filters the records based on a given condition. Say one would like every country with a population greater than 10.000, they would have a selection as $\sigma_{population>10000}$
- *Project (Π)* The project operator, as the name implies, projects the data onto the desired projection. As an example using the countries database, say we are just interested in the country names. Then we can use a projection to just retain that column as $\Pi_{countryname}$. Another use case of the projection can be where some columns in the data are not in a correct format (say minutes instead of hours). We can then do a projection to retrieve the amount of hours.
- *Join ($\bowtie$)* Another important operator is the (inner) join operator. As the name implies this operator represents the join between two data results given a condition. Say we would like to see the languages spoken in all countries we could join the country and language tables like: $\bowtie_{country.language\_id=language.id}$.

Now, lets say we have the following database setup:

Name	Age	ID

Person table

S_ID	P_ID

Student table

In this database, there are teachers and students and both of them are persons. We are now interested in finding the names teachers with an age greater than 26 which are also themselves students. This would translate to the following query:

LISTING 3.1: Example query

```
SELECT teacher.name from teacher
JOIN person ON teacher.person_id = person.id
JOIN student ON student.person_id = person.id
WHERE teacher.age > 26
```

This query can then be transformed in the following logical operator tree (T):

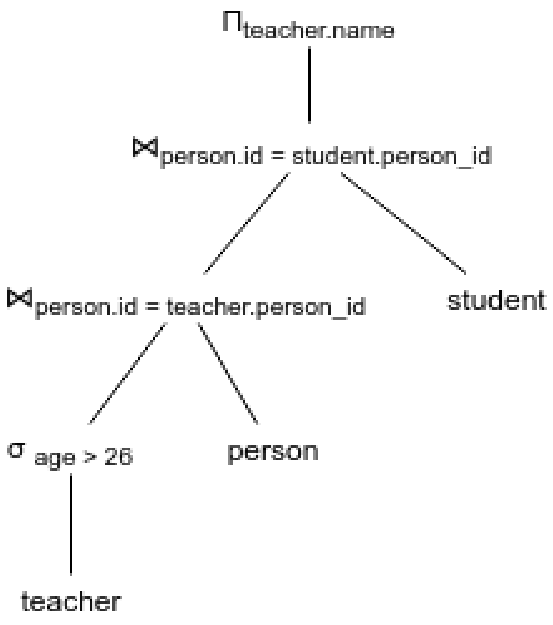
Π teacher.name
⋈ person.id = student.person_id
⋈ person.id = teacher.person_id
student
σ age > 26
person
teacher

FIGURE 3.1: The logical operator plan of query 3.1

As we can see, the selection on teacher.age has been pushed down to the teacher table. Then teacher is joined with person and after that with student. Finally, the teacher.name is projected.

From this tree we can identify 4 possible subexpressions: $t_1, t_2, t_3, t_4 \subseteq T$. One is the selection on teacher.age. Another is the join on person. To get the third subexpression we also join student with teacher and person. And finally the last subexpression is the original query. These 4 subexpressions are identified more clearly in figure 3.2.

Or, more formally, for a given logical operator tree T, every subtree $t \subseteq T$ is considered a subexpression.

T_ID	P_ID

Teacher table

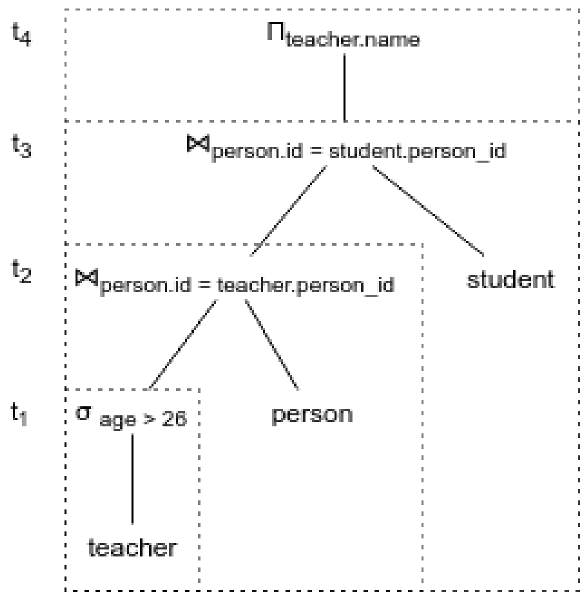
t4
$\Pi_{teacher.name}$
t3
$\bowtie_{person.id = student.person\_id}$
t2
$\bowtie_{person.id = teacher.person\_id}$
student
t1
$\sigma_{age > 26}$
person
teacher

FIGURE 3.2: The identified subexpressions in the logical operator tree 3.1

3.2 Query workload

A query workload Q describes a collection of queries $q_1, q_2, \ldots, q_n \in Q$ to be issued to the database.

3.3 Query rewriting

Query rewriting describes the process of finding elements in the logical plan of the query and replace them with more efficient elements. Query rewriting allows for another way for the DBMS to optimize the query. There are 2 ways to optimize the query via query rewriting: Partial rewriting and exact rewriting.

3.3.1 Partial rewriting

Partial rewriting involves rewriting an expression into another expression that might not fully contain the original expression or contains more information than the former expression. In the case of the former, we would be looking at a union rewriting between the new expression and the expression representing our missing data. In the case of the latter, we would be looking at adding some transformation rules (for instance, add an extra selection) to match the original data request.

Goldstein et al. 2001 have created a method to achieve partial and equivalent rewriting by analyzing the structural information of the query plan. Their implementation however, focuses only on finding a rewriting for SPJA-queries.

3.3.2 Equivalent rewriting

With equivalent rewriting we try to find subexpressions that produce the same output but are less costly to produce. One such example is by replacing costly subexpressions with a tablescan of a materialized view (which will be used for the experiment). An easy way to achieve equivalent rewriting for single expressions is by storing each query by a key. Since equivalent queries can be expressed differently we would have to parse the logical plan of each query to see which queries are equivalent.

To efficiently find an equivalent rewriting, we first need to know the hashcode of each subexpression in a logical plan. Let's assume that hashing a node in the tree takes O(1), and there are n nodes in the tree. This would mean that finding the hashcode of a logical plan (and thus also each subexpression) takes O(n) time.

Finally, for each subexpression in the logical plan we can do an efficient lookup using the hashcode to find equivalent nodes and replace the original nodes in the logical plan with the more efficient equivalent code.

As an example, say we are using the student database as described in section 3.1. However, the database now also includes a materialized view joining person and student. Just like in the previous section, the database got issues the following query:

```
SELECT teacher.name from teacher
JOIN person ON teacher.person_id = person.id
JOIN student ON student.person_id = person.id
WHERE teacher.age > 26
```

And without any query rewriting, the logical plan would not alter from 3.1.

By rewriting the query using the materialized view, we can avoid having to rejoin those tables allowing a more efficient logical plan. The result of the rewriting can be found in figure 3.3

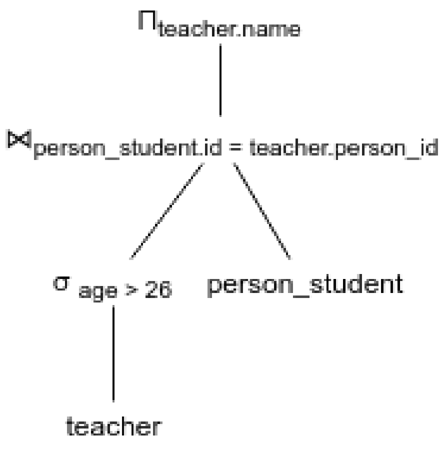

FIGURE 3.3: Equivalence rewriting using the person_student materialized view of plan 3.1

3.4 Knapsack problem

One of the more well-known optimization problems is the knapsack problem. The knapsack can be described as: given a set of items each with a weight and value, determine the set of items to include in the knapsack maximizing the total value without exceeding the total weight limit.

There are many variations for the knapsack problem, including but not limited to the 0/1-knapsack problem and the quadratic knapsack. In this section we will cover the 0/1-knapsack and in the next section we will cover the quadratic knapsack.

The 0/1-knapsack, as the name implies, is a variation of the knapsack problem where each item can either be included or not. There are no partial items in the knapsack. The problem can be defined as follows: Given items $x_1, x_2, \ldots, x_n \in X$, each with value $v_1, v_2, \ldots, v_n \in V$ and weight $w_1, w_2, \ldots, w_n \in W$. Find a set of items $x \in X$ satisfying the following linear equation.

maximize $\sum_{i=1}^{n} v_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0, 1\}$

3.4.1 Brute force

One of the more pragmatic approach is to just brute force the solution. This approach comes down to iterating over the power set of items and finding the maximum value set of items without exceeding the weight limit. A brute force algorithm for the 0/1-knapsack problem can be defined as follows: Let $P(X)$ be the function of returning the power set of items $x_i \in X$. Then the following linear equation would describe the brute force approach.

$$\operatorname*{argmax}_{x \subseteq P(X)} \sum_{i=1}^{n} v_i x_i$$
$$\sum_{i=1}^{n} w_i x_i \leq W$$
$$x_i \in \{0, 1\}$$

This algorithm will always find the best possible solution given enough time. Unsurprisingly, the runtime of this algorithm is not to be favored. Since we would be iterating over the power set of n items, giving 2^n sets to iterate over, we should be expecting a runtime of $O(2^n)$.

3.4.2 Greedy

A more computational efficient algorithm, although not completely accurate, is the greedy approximation algorithm. The basic idea for the greedy algorithm is finding the items with the highest value per weight category and putting them in the knapsack until no more items can fit in the knapsack. Like most greedy algorithms, this approach seems very intuitive but does not always provide the best solutions.

For the runtime analysis of this greedy algorithm we are upper bounded by the sorting of each item. Since sorting takes on average an runtime of O(n log n), the runtime of this algorithm is O(n log n).

Since the greedy algorithm is an approximation, it is important to know its approximation ratio to make sure that the solutions, at worst, don't deviate too much from the optimal solution. For the approximation of the algorithm, lets say for some small $\epsilon > 0$ there are 3 items:

Algorithm 1 Greedy approach for the 0/1-knapsack

```
 1: $X \leftarrow$ all available items $x_i$, each with weight and value
 2: $X \leftarrow sort(X)$                    ▷ sort each item by $\frac{x_i.value}{x_i.weight}$ descending
 3: $W \leftarrow$ capacity of the knapsack
 4: $S \leftarrow \emptyset$
 5: for all $x_i \in X$ do
 6:     if $x_i.weight \leq W$ then
 7:         $S \leftarrow S + x_i$
 8:         $W \leftarrow W - x_i.weight$
 9:     end if
10: end for
11: return $S$
```

- item 1 with value $w + 2\epsilon$ and weight $w + \epsilon$
- item 2 with value w and weight w
- item 3 with value w and weight w

Given is also that the size of the knapsack is $2w$. The greedy algorithm will select item 1 and not find any more items that can fit in the bag for a total value of $w + 2\epsilon$. The optimal solution consists of item 2 and item 3 together, for a total value of $2w$. As $\epsilon \rightarrow 0$, we find that the approximation ratio
$R_a(I) = \frac{OPT(I)}{A(I)} = \lim_{\epsilon \rightarrow 0} \frac{2w}{w+2\epsilon} = 2$. In other words, in the worst case scenario the greedy algorithm would return results half of the optimal value we can pick.

3.4.3 Dynamic programming

A third algorithm for solving the knapsack algorithm is a dynamic programming algorithm (Kleinberg and Tardos, 2013). The idea behind dynamic programming is that sometimes one can find repeating subproblems in the original problem and by computing and memorizing those subproblems one can more efficiently compute the solution of the original problem.

For the knapsack problem the dynamic programming approach goes over each individual item in sequence given a weight limit. Say there are n items and a maximum weight of w. For each item, the algorithm considers whether, given the current weight, it is more beneficial to not include item i or to include it and reduce the weight limit, and then goes on to the next item until all items are considered.

$$M[i,j] = \begin{cases} 0, & \text{if } i = 0 \\ 0, & \text{if } j \leq 0 \\ \max M[i-1,j], M[i-1,j-w_i] \end{cases}$$

In the final case of the function we can see that it considers two options, not including item i (thus not influencing the weight) or including item i and reducing the weight.

Let's assume all n items have equal weight. Now consider two scenarios, in the first scenario we do not include the first item but include the second item. In the second scenario, we include the first item but do not include the second item. In both scenarios we encounter the same subproblem for the remaining n-2 items.

Using the dynamic programming approach, we would only have to compute this subproblem once.

Since we would need to consider at most for each item and for each weight, the runtime of the dynamic programming is O(nW). Since the runtime depends on the value of W, this algorithm is considered pseudo-polynomial.

Outside of a runtime complexity of O(nW), this algorithm also has a space-time complexity of O(nW) since we need to be able to store each intermediate result.

3.5 Quadratic knapsack

Whereas with the traditional binary knapsack problem we had a linear optimization problem, the quadratic knapsack problem aims to optimize a Quadratic objective function (Pisinger, 2007). The idea behind the quadratic knapsack problem is that in some cases we could add more value to including an item if another item is added to the knapsack. Let's say we are packing for our holiday, and we take a few books with us. A practical example of adding more value to other items would then be that by taking books with us, we are also more inclined to take our reading glasses with us.

The quadratic knapsack problem can be defined as follows. Assume there are n items and for each item i there is a positive weight w_i and value v_i. We also have an n x n non-negative matrix P where each item p_{ij} indicates the value gained by having both items i and j included. Then the binary quadratic knapsack can be defined as:

$$\textit{Maximize} \sum_{i=1}^{n} v_i x_i + \sum_{i=1}^{n} \sum_{j=1,ji/}^{n} p_{ij} x_i x_j$$

$$\textit{Subject to} \sum_{i=1}^{n} w_i \leq W$$

3.6 Candidate view selection

This paper will describe candidates as suitable views/expression to materialize in order optimize the performance of the query workload. After identifying these candidates, using either partial (section 3.3.1) or equivalent rewriting (section 3.3.2) we can substitute these candidates into the original operator tree resulting in a more optimal tree. After identifying these candidates, we can analyze them with regard to the query workload to find which candidates would be the most optimal to materialize to disk and use as a sub solution for the current queries.

The two main approaches discussed here are the view-selection approach and the subexpression selection approach. For performance reasons, we will focus on the second approach in this thesis.

3.6.1 View selection

The aim of view selection is to find a view that can function as a suitable candidate for optimizing a query workload. Unlike subexpressions, views are not limited to the query input and can therefor contain any section of the data that is most suitable for the query workload. The problem of finding candidate views is also called the view selection problem and can be defined as: given a database schema and a query workload, find a set of views to materialize in order to improve the query workload (Mami and Bellahsene, 2012).

3.6.2 Subexpression selection

As mentioned earlier, subexpression selection focuses on finding candidates out of the set of subexpressions. The main difference between view selection and subexpression selection can be found in the solution space. With view selection, we have the entire solution space to reason about whether a view can be a worthy candidate. With subexpression selection, we are limited by the subexpressions found in our data. The advantage of this limited search space is that finding a suitable candidate is computationally a lot easier. The trade off, of course, is that the found candidate might not be the most optimal candidate we can find.

3.7 The experiment

In the experiment the focus lies mainly on finding subexpressions in a similar repetitive query workload. To achieve this, we will be analyzing previously executed queries and find common subexpressions among these queries. The assumption is then that since the workload consists mostly of similar repetitive queries, it should be possible to find common subexpressions among a new workload by analyzing the previous query workload.

The hypothesis is that by finding and materializing subexpressions in a previous workload, we can improve the performance of queries in a new workload.

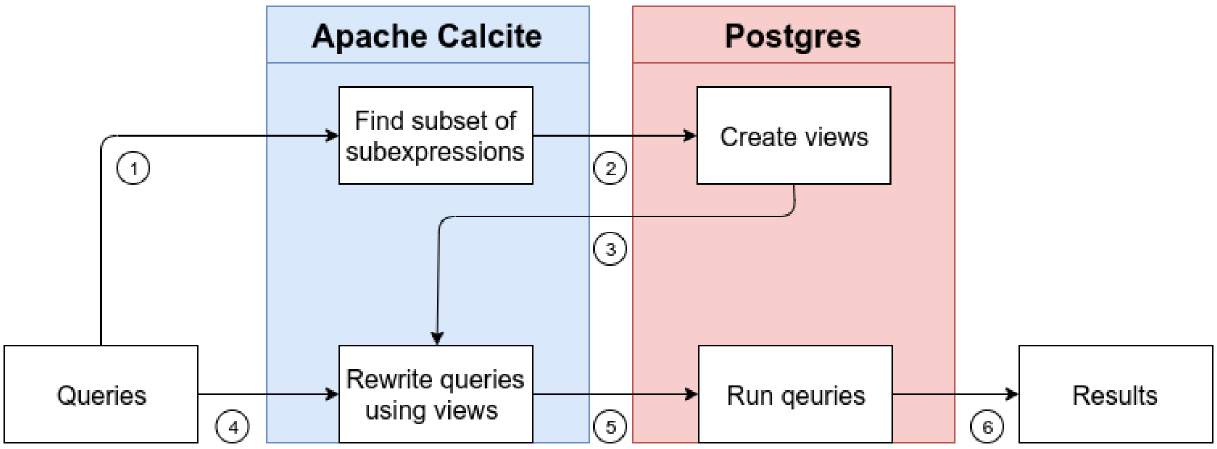

FIGURE 3.4: General pipeline setup of the experiment

A general pipeline of the experiment can be found in figure 3.4. This pipeline describes the general process of the experiments and we can see 6 labelled edges. The process goes as follows: first in edge 1 we send the queries to Apache Calcilte to extract and find a subset of meaningful subexpressions to materialize. Then in edge 2 we ask Postgres to create the views containing the subset of subexpressions. After these subexpressions are materialized we let Apache Calcite know that these views exists in the database, represented by edge 3. After we let Apache Calcite know about these newly created views we can start rewriting the queries, denoted by edge 4. After rewriting the queries we send the rewritten queries to Postgres to execute them (5). Finally, in edge 6 we have the query results.

This work can be especially useful in applications where these simple independent queries are the norm of the workload, which can be quite common among a plethora of databases. An example could be checking if a certain product is still in stock and, if needed, resupply said product in an internal product database.

3.7.1 Finding subexpressions

Say there are 2 queries q_1 and q_2 consisting of $\{s_1, s_2, s_3\}$ and $\{s_2, s_3\}$ respectively and s_1, s_2 and s_3 have the same profit and cost. In this example, we would never consider materializing s_1 as our first query since it only benefits q_1 whereas s_2 and s_3 benefits both q_1 and q_2.

To define a candidate subexpression we need to look at the logical plan of a query. Let $q \in Q$ be a query with a corresponding logical plan t. Any subtree t' of t, including t, is then considered as a candidate subexpression.

Then, for each candidate subexpression we compute the cost and profit of materializing said subexpression. The cost of materializing a subexpression will be defined as the size of the materialized subexpression. For the profit of a subexpression, we have to look at all the queries in the query workload which are using said subexpression. Lets define the utility $u_{ij} \in U$ as how much query q_i is benefited from materializing subexpression s_j. The utility of a query q_i for subexpression s_j is computed as follows:

$$U_{ij} = C_c(q_i) - C_{acc}(q_i)$$

Where $C_c(q_i)$ defines the cost of evaluating query q_i without materializing subexpression s_j and $C_{acc}(q_i)$ the cost of evaluating query q_i while having access to a materialization of subexpression s_j.

To compute the total profit of materializing subexpression s_j we would need to sum all utilities provided by subexpression s_j.

$$s_{j.profit} = \sum_{q_i \in Q} U_{ij}$$

Finding the optimal set of subexpressions to materialize can be proven to be np-hard. After materializing one subexpression, every subexpression that uses this subexpression will have its cost and profit reduced (since they can now use the new subexpression). Lets define the profit matrix P where each element $P_{ij} \in P$ represents the effect the materialization of subexpression i has on the profit subexpression j. Using this definition, we can reduce the problem of finding the optimal set of subexpressions to the quadratic-knapsack problem, for which the optimization problem is proven to be np-hard.

Therefor, instead of finding an exact solution an approximation will be used using a greedy approach. Given there is $s_1, s_2, s_n \in S$ where S is the list of all candidate subexpression, each with a profit and size s_{profit} and s_{size} respectively, find the next subexpression with the highest factor of $\frac{s_{profit}}{s_{size}}$ which can still fit according to the memory constraint.

After a new candidate subexpression is identified, the profit of the other subexpressions need to be updated. The update process is needed since after a new candidate subexpression is selected, other subexpressions might get their value reduced since they can now use the selected subexpression as their intermediate result instead of recalculating the results of the subexpression.

The pseudocode of the algorithm can be found in algorithm 2

Algorithm 2 Finding an optimal set of candidate subexpressions to materialize

```
1: $Q \leftarrow$ Query workload
2: $S \leftarrow$ parseQueries($Q$)
3: $cap \leftarrow$ CAP
4: $newElement \leftarrow$ True
5: $selected \leftarrow \emptyset$
6: while $newElement = true$ do
7:     $newElement \leftarrow$ False
8:     $candidate \leftarrow s_{-1}$                    ▷ $s_{-1}$ has zero profit and a size of 1
9:     for all $s_i \in S$ do
10:        if $\frac{s_i.profit}{s_i.size} > \frac{candidate.profit}{candidate.size}$ AND $s_i.size \leq cap$ then
11:            $candidate \leftarrow s_i$
12:        end if
13:    end for
14:    if $candidate \neq s_{-1}$ then
15:        $selected \leftarrow selected + candidate$
16:        $cap \leftarrow cap - s_i.size$
17:        $newElement \leftarrow$ True
18:        $S \leftarrow$ update($S$, $candidate$)
19:    end if
20: end while
21: return $selected$
```

3.8 Summary

In this chapter we have discussed the general methodology as well explained some of the terminology used throughout the thesis. In section 3.1 we have looked at the definition used for queries and subexpressions while also looking at how these 2 terms are related. To further expand on these defintions we also briefly looked at the definition of a query workload in section 3.2.

We have also taken a more in depth look into query rewriting in section 3.3. We found that query rewriting can be divided into two different approaches: partial rewriting and equivalent rewriting. In section 3.3.1 we found that a view does not always contain the exact information. Sometimes you might have to add information using union rewriting or remove information by filtering. This is called partial rewriting. Then in section 3.3.2 we discussed the situation where the view exactly describe a subexpressions and we can substitute a tablescan of that view with the subepressions, which is called equivalent rewriting.

Then, in section 3.4 we briefly discussed the knapsack problem since we will also rewrite our problem as a knapsack problem. We briefly discussed three popular algorithms to solve the knapsack problem: brute force, greedy and via dynamic programming. Since our problem has the property that some subepressions can have effect on other subepressions, we also briefly went over the quadratic knapsack problem in section 3.5.

In section 3.6 we once again briefly covered the candidate view selection problem, divided into two subsections: view selection in 3.6.1 and subexpression selection in 3.6.2.

Finally, in section 3.7 we have taken a look at the general pipeline. After discussing the general pipeline we then dived deeper into the general selection and evaluation process of the subexpression in section 3.7.1

Chapter 4

Implementation

As mentioned previously, the main goal of this thesis is to accelerate the processing of queries by materializing common subexpressions. To test the hypothesis I have set up an IMDB-database using Postgres 10.8. Since Postgres on its own is unable to rewrite queries using materialized subexpressions, the system also makes use of Apache Calcite 1.19 These experiments will be run on a Dell XPS 15 - 9570 laptop with an Intel I7-8750H processor on an SSD using 16 (2*8) gigabytes of RAM.

This chapter will be structured as follows. Firstly the IMDB-database will be explained in more detail in 4.1. Then in 4.2 we will discuss how the queries used in the experiment are generated. Section 4.3 will discuss the evaluation of the cost, profit and size of the different subexpression. Finally, in 4.4 the difficulties with regard to working with Apache Calcite will be mentioned as well as some encountered bugs in the rewriter of Apache Calcite.

4.1 Database setup

The experiment will be conducted using the IMDB-database. The schema of this database can be seen in figure 4.1

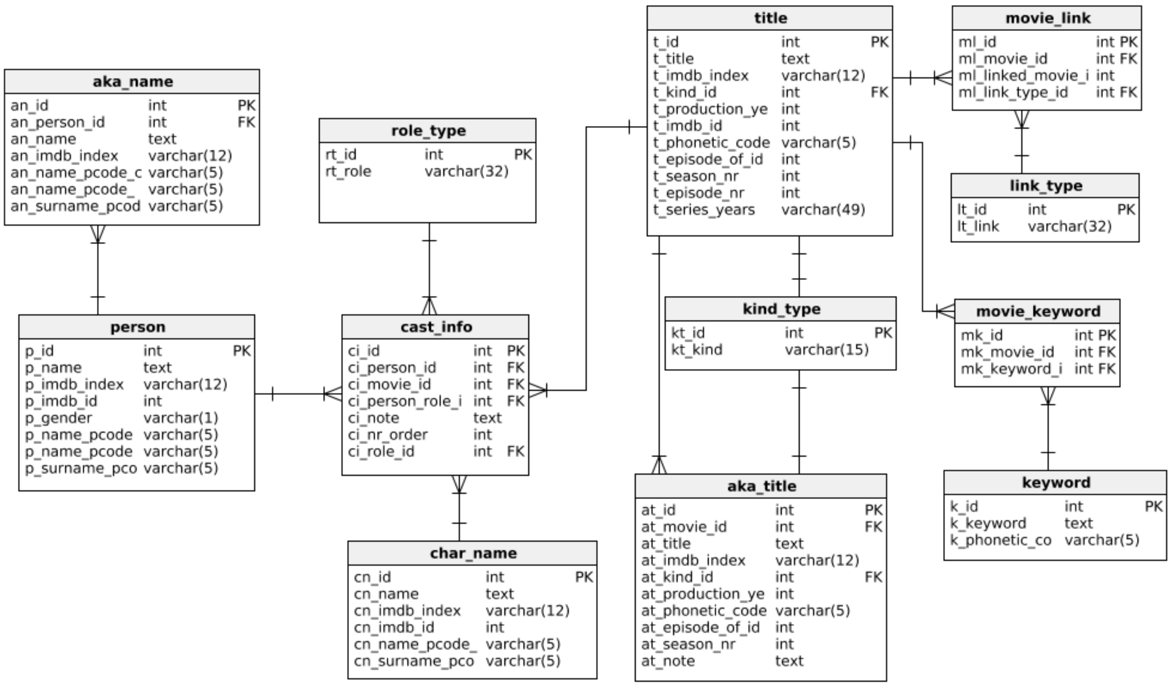

FIGURE 4.1: Schema of the IMDB-database.

Table	Amount of records	Size per record	Size in megabytes
movie_link	2581826	16	41.31
aka_title	526114	84	44.19
aka_name	1130179	41	46.34
link_type	18	86	0.01
cast_info	62039343	42	2605.65
movie_keyword	7415173	12	88.98
kind_type	12	52	0.01
keyword	236350	26	6.15
role_type	12	86	0.01
title	4626969	61	282.25
person	6179000	42	259.52
char_name	4313697	75	323.53

TABLE 4.1: Size of the different tables in the database.

In table 4.1 we can find the amount of records and the size of the different tables. For each of these tables, we can also find the statistics of a subset of columns (amount of unique values, the minimum value and the maximum value) in table 4.2

4.2 Synthetic query generation

Since no query workload dataset could be easily found, I have made the decision to generate them. The queries generated for the experiment are simple SPJA (select project join aggregate) queries. The reason the limit ourselves to just SPJA-queries is because in a real workload these queries will be the majority of your workload. For aggregates we will mainly be focusing on the count aggregate. These simple queries will keep the query generation step simple as well as being the type most benefited by the approach taken. For generating these queries, the generator will take one of the following templates at random:

- select * from *RANDOM_TABLE*
- select *RANDOM_COLUMN* from *RANDOM_TABLE*
- select count(*RANDOM_COLUMN*) from *RANDOM_TABLE*
- select * from *RANDOM_TABLE* where *RANDOM_COLUMN* = *RANDOM_VALUE*

. Where *RANDOM_COLUMN* is a randomly selected column of the selected tables and *RANDOM_VALUE* is random value of *RANDOM_COLUMN*. *RANDOM_TABLE* can either be a randomly selected table from the database 4.1, or a join between these tables. To get some idea about the sizes of these joins, I have analyzed a handful of these joins. These results can be found in table 4.3

4.3 Defining cost, profit and size

Using Postgres, we can easily find the cost and size of a subexpression. Say, we have a subexpression t, we can convert this into a relational query by wrapping it into a select-command as: select * from t.

Table	Column	Unique values	Min	Max
aka_name	an_person_id	724599	5	6232404
aka_name	an_name	1002881	?	Þura
aka_name	an_id	1130179	2	1311422
aka_title	at_movie_id	300749	100210	4716411
aka_title	at_kind_id	7	1	8
aka_title	at_id	526114	1793	563154
aka_title	at_episode_nr	237	1	13913
aka_title	at_season_nr	56	1	111
aka_title	at_title	456166	<—>	þyÞÁ 2
aka_title	at_episode_of_id	2371	1798	563155
aka_title	at_production_year	139	1874	2024
cast_info	ci_role_id	11	1	11
cast_info	ci_movie_id	4139462	100000	4720644
cast_info	ci_person_role_id	4267814	1	4313697
cast_info	ci_person_id	6179000	1	6232410
cast_info	ci_id	62039343	1	63372001
char_name	cn_name	4312673	ʻ	Þýskur
char_name	cn_id	4313697	1	4313697
keyword	k_id	236350	1	236350
keyword	k_keyword	236350	7	zz-top-impression
kind_type	kt_id	12	1	12
kind_type	kt_kind	11	aap	video movie
link_type	lt_id	18	1	18
link_type	lt_link	18	alternate lang.	version of
movie_keyword	mk_id	7415173	41941	7475778
movie_keyword	mk_movie_id	718489	100027	4720644
movie_keyword	mk_keyword_id	235564	1	236350
movie_link	ml_movie_id	325776	2	4720620
movie_link	mk_linked_movie_id	325775	2	4720922
movie_link	ml_link_type_id	16	1	17
movie_link	ml_id	2581826	1	2581826
person	p_id	6179000	1	6232410
person	p_gender	2	f	m
person	p_name	5110061	0010x0010	þumlungur, Jón
role_type	rt_role	12	actor	writer
role_type	rt_id	12	1	12
title	t_series_years	1652	????	2022-????
title	t_production_year	144	1874	2115
title	t_episode_of_id	98348	1779	4726969
title	t_title	2507170	<—>	Þyngdarafl
title	t_id	4626969	100000	4726969
title	t_kind_id	7	1	8
title	t_episode_nr	15138	1	91334
title	t_season_nr	152	1	2015
title	t_season_nr	152	1	2015

TABLE 4.2: Column statistics for a subset of columns in the database.

Table	Amount of records	Size per record	Size in megabytes
aka_title ⋈ kind_type	526114	136	71.55
cast_info ⋈ char_name	28749612	117	3363.70
cast_info ⋈ role_type	62039343	128	7941.04
person ⋈ aka_name	1130179	83	93.80
person ⋈ cast_info	62039343	84	5211.30
title ⋈ aka_title	526114	145	76.29
title ⋈ cast_info	62039343	103	6390.05
title ⋈ kind_type	4626969	113	522.85
person ⋈ cast_info ⋈ title	62039343	145	8995.70
title ⋈ movie_link ⋈ link_type	2567141	163	418.44

TABLE 4.3: Size of different analyzed joins in the database.

After converting a subexpression back into a relational query, we can simply do an explain command on Postgres to find the total cost and the size. For instance, say we have the following query, which is a relational query created from the subexpression Join(aka_title, title):

```
explain     select *
            from aka_title at
            join title t
            on at.at_movie_id = t.t_id
```

This will return the following top-level result:

```
"Merge Join  (cost=20.02..207189.44 rows=526114 width=145)"
```

For the cost, I would always assume worst case, which in this case is 207189.44 units of an arbitrary computation unit. For the size we have a row count estimation of 526114 rows each with a width of 145 bytes, bringing the size to 76286530 bytes or 76.3 Mb for short.

To define the profit, I first need to know the cost of scanning a materialized subexpression and then subtract that from the cost of evaluating a subexpression.

Algorithm 3 Computing the cost, profit and size of a subexpression for 1 query

```
1: $s \leftarrow$ subexpression
2: $q \leftarrow$ createQuery($s$)
3: $explain \leftarrow$ explain($q$)
4: $size \leftarrow explain.rows * explain.width$
5: $blocks \leftarrow$ getBlocks($size$, $blocksize$)
6: $cpuCost \leftarrow coefficient * explain.rows$
7: $scanCost \leftarrow blocks + cpuCost$
8: $profit \leftarrow explain.cost - scanCost$
```

For Postgres, the scan cost consists of 3 components: the IO-cost, the cpu per tuple cost, and the cpu per operator task. The IO-cost is equal to the amount of blocks required. the cpu per tuple cost is calculated as a coefficient times the amount of tuples and the cpu per operator task is also defined as a coefficient times the amount of tuples *costsize.c*. Using Postgres default values we can find that the sum of the coefficients for both cpu operator and cpu per tuple cost is equal to 0.0125 *cost.h*. Furthermore, the default blocksize used by Postgres is 8192. Substituting these values into the algorithm 3 we can find the cost, profit and size for a subexpression with regard to a single query. To find the total profit of materializing a subexpression we would have to count the amount of queries using said subexpression and multiply the profit of the subexpression with that amount.

4.4 Issues with Apache Calcite

One of the main challenges of the thesis has mostly been focused on the integration between Apache Calcite and Postgres. Apache Calcite is an impressive system with a lot of bells and whistles. The main problem I found with using Apache Calcite is that it can sometimes seem over engineered without proper documentation. For instance, were we to try out Apache Calcite ourself and follow the example on their main web page we will find that the example will not work due to case sensitivity (*table not found with apache calcite*) My experience with Apache Calcite was that if I wanted to use Apache Calcite to do some basic operations it can work great. However, the moment I wanted to do something a bit more advanced I stumbled upon a lot of poor or sometimes even missing documentation issues, with the only support being the developer mailing lists since the presence of Apache Calcite on more traditional support forums like *StackOverflow* has been really low.

At first, I wanted to use Apache calcite as a communication layer between the application and the Postgres Database 4.2. The idea was that the application receives a query, then sends it to Apache Calcite, which then transforms it into a query using the available materialized views and then sends it to Postgres. I quickly found out that this is not the way to go forward. Not only am I not really able to analyze the logical query operator tree this way, but Apache Calcite is also not able to process most of the queries. Apache Calcite processes the queries in main memory, and asking Apache Calcite to join two large tables thus results in an out of memory exception. Furthermore, another important aspect of the system is to be able to create materialized views from the different subexpressions. Apache Calcite advertises on their web page that it is capable of creating materialized views, but practice showed quite the opposite. Before a materialized view is even created Apache Calcite tries to occupy it with data, meaning that it asks the database to insert rows in a table that doesn't even exists.

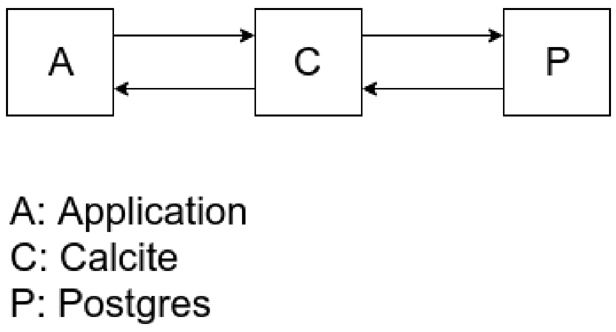

FIGURE 4.2: Communication from application to Apache Calcite to Postgres and back.

So, instead of using Apache Calcite as a layer between the application and the Postgres database I will be using Apache Calcite to rewrite queries 4.3. In short, the application receives a query and then sends it to Apache Calcite to rewrite those queries to use materialized views and then send those rewritten queries to Postgres.

Id	Name
1	Klaas
2	Jan
3	Piet

TABLE 4.4: Person Table A

Id	Year of birth
1	1980
2	1990
3	2000

TABLE 4.5: Person table B

In this setup, Apache Calcite's main purpose is to rewrite those queries to use materialized views. Any other optimization should be avoided since Postgres will have its own optimization process and we would like to avoid doing any unnecessary optimization by Apache Calcite. The advantages of using Apache Calcite this way is that it allows me to access the internal logical nodes of Apache Calcite to find subexpressions and therefor materialized views. The main challenge this approach has is how to accurately translate the logical node tree from Apache Calcite back into a string representation for Postgres. Luckily, Apache Calcite does provide support to translate these trees back into string queries. Occasionally though, it does fail to retrieve sensible output for Postgres to parse.

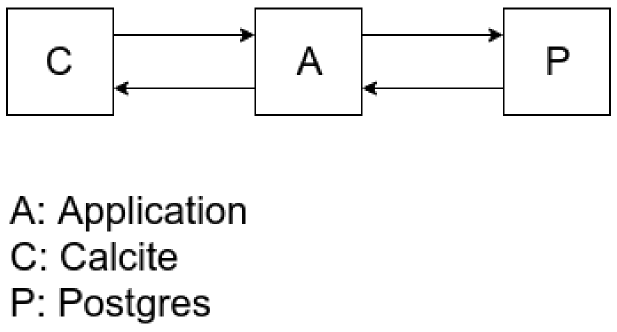

FIGURE 4.3: Communication from application to Apache Calcite to application to Postgres and back.

Another issue found while integrating these two systems is the differences in acceptable query languages. Apache Calcite needs explicit mentions of the schemaname as well as a case-sensitive tablename description whereas Postgres is a bit more flexible in that regard.

Finally, the biggest challenge I faced with incorporating Apache Calcite for rewriting concerned the union rewriting of the heuristic planner. There is a bug in the heuristic planner of Apache Calcite where, during rewriting, it creates a circular rewriting. To illustrate, say I have 2 tables with person information: A and B. Table A (4.4) contains the id of a person, and their names. Table B (4.1) contains the id of a person, and their year of birth. Also, there is a materialized view mv1 (4.6) containing the join between A and B where their year of birth < 2000.

The bug occurred when we wanted to join A and B. First, Apache Calcite creates a logical operator tree for this join according to figure 4.4.

Id	Name	Year of birth
1	Klaas	1980
2	Jan	1990

TABLE 4.6: Materialized view mv1

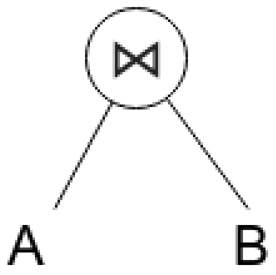

FIGURE 4.4: Logical operator tree of a join between table A and table B.

To optimize this tree Apache Calcite then transforms this tree into a union between mv1 and the join between A and B where year of birth $\geq$ 2020. This results in following tree (figure 4.5)

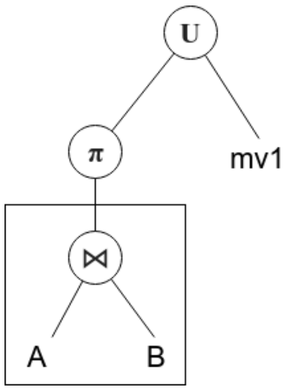

FIGURE 4.5: Logical operator tree of a join between table A and table B after rewriting.

In the next step, the optimizer steps through the tree and encounters another join between tables A and B. This is where the bug happens. The rewriter does not take into consideration this part of the tree is already optimized and optimizes it again resulting in another union. This cycle will repeat indefinitely until a StackOverflow happens.

Internally, what happens is that the heuristic rewriter uses a DAG representation of the tree. When the rewriter passes a node it keeps a reference of this node as a parent. When it wants to do a rewrite it checks if the parent is not the same node to avoid these cycles. However, what this rewriter does not take into consideration is what happens when an indirect parent is of the same node which also causes these cycles. To resolve this issue I had to make sure that before every optimization step check if there is any parent of the same node and if so, do not rewrite this node.

4.5 Summary

In this section we have looked at the implementation details. First, in section 4.1 we have taken a look at the database setup used for the experiments. Then, since we had trouble finding a suitable and representative query dataset we decided to generate these queries ourself which we discussed in section 4.2. In order to decide whether a subexpression is worth materializing we need to know about its size and potential gain. This was discussed in section 4.3.

Finally, we also looked at the difficulties we faced while working with Apache Calcite in section 4.4. Most notably, the lack of clear and correct documentation as well as the lack of a proper support platform to ask questions. Furthermore, we also discussed a bug in the union rewriting in Apache Calcite and how we were able to resolve this.

Chapter 5

Evaluation

In this chapter we are going to evaluate the results obtained from the experiments. First, in 5.1 we will discuss the general setup taken for the experiments. Then in 5.2 we are going to look at the difference in runtime between running the queries without materialized subexpression and running the queries with materialized subexpressions. After that, in 5.3 we are going to look the amortization period, the amount of queries it takes to see a positive effect when materializing subexpressions. Finally, in 5.4 we will dive deeper into the rewriter of Apache Calcite and how it scales with regard to increasing amount of queries in the query workload and increasing amount of materialized views.

5.1 Experiment setup

The experiment will be executed with a query set of 400 queries in two phases. In the first phase there will be no separation of training and test set, meaning that the system will be in the most optimal situation (since it is testing on the same data from which it derived favorable subexpressions). In the second phase there will be a separation of training and test set using cross validation of 4 iterations using 100 queries as test set en 300 queries as training set (Koutroumbas and Theodoridis, 2008).

The idea is that by increasing the amount of memory allowed to be used for materializing subexpressions one can increase the performance of the query workload by materializing more subexpressions or perhaps different ones with more profit.

During the experiment I will measure the time it takes with and without materialized subexpressions. I will also measure how much time it takes to materialize a subexpression and then run the workload to see if materializing is worth it. Finally, I will be measuring the total time it took to run a query and the time it took Postgres to complete a query. Measuring both times is necessary since it allows us to find the amount of time it takes Apache Calcite to find a rewriting using the materialized subexpressions.

5.2 Query runtime

For the results I have captured the cost and time of the query using Postgres *explain analyze* command, as well as measuring the total time (including Apache Calcite's rewriting time) using Java's *System.currentTimeMillis()*. Using the same query workload for training, as well as testing, a significant reduction in execution time can be noted (with around 50% reduction in execution time starting from 20 gigabytes of allocated memory) 5.1.

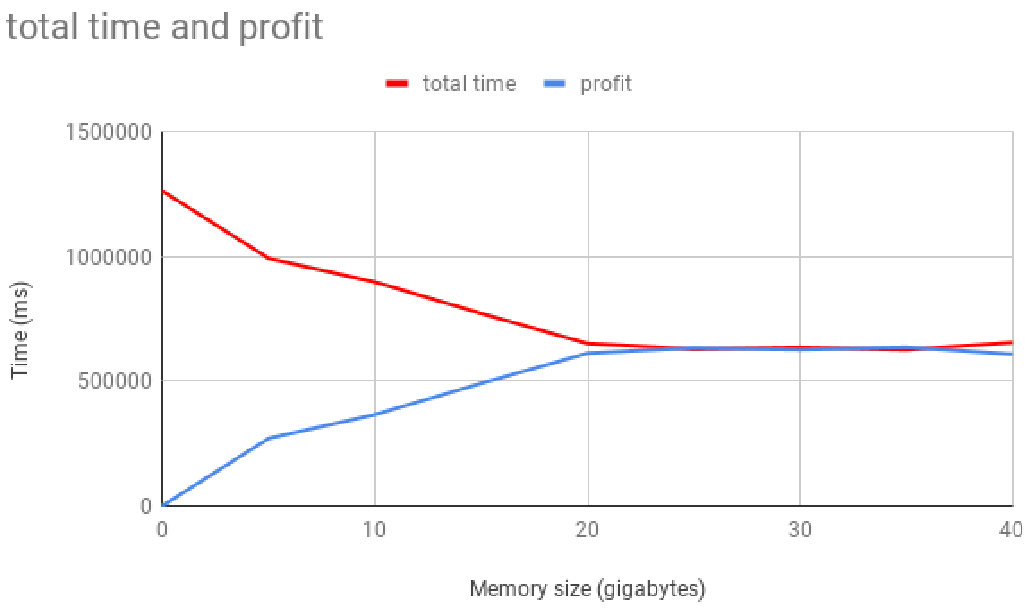

FIGURE 5.1: The total time of executing the queries and the profit (time saved) by using materialized subexpressions using the same training and test set.

These results, however, are only true in the best case scenario where the query workload is equal to the queries seen by the system. For a more accurate result I have repeated the experiment but instead done 4 iterations using a split of 300 queries for training and 100 for testing, as described in chapter 3. These results are then summed up to allow a comparison between the optimal setup and a split setup. The total time and the profit gained using this method can found in figure 5.2 A comparison between a more realistic training and test split using cross validation, and the first scenario where no such split is used can be found in figure 5.3

By measuring both the query time, and the total time it is be possible to see the impact Apache Calcite has on finding a rewriting. These measurements can be found in figure 5.4.

These results show us that by materializing subexpressions and rewriting queries to use these subexpressions we can expect quite a jump in performance without influencing the planning time much using similar queries. The downside however is that this performance can only be achieved if the subexpressions are either already materialized, or the query workload is of such significant size that the cost of materializing the candidates as well as running the workload is less than running the workload.

An example where this is not the case can be seen in figure 5.5. Here we can clearly see that materializing the subexpressions and running the workload costs way more than just running the workload without materialization.

5.3 Amortization period

It is important to know at which point the workload is of such sizes that materializing subexpressions and running the workload costs the same as just running the workload, the amortization period.

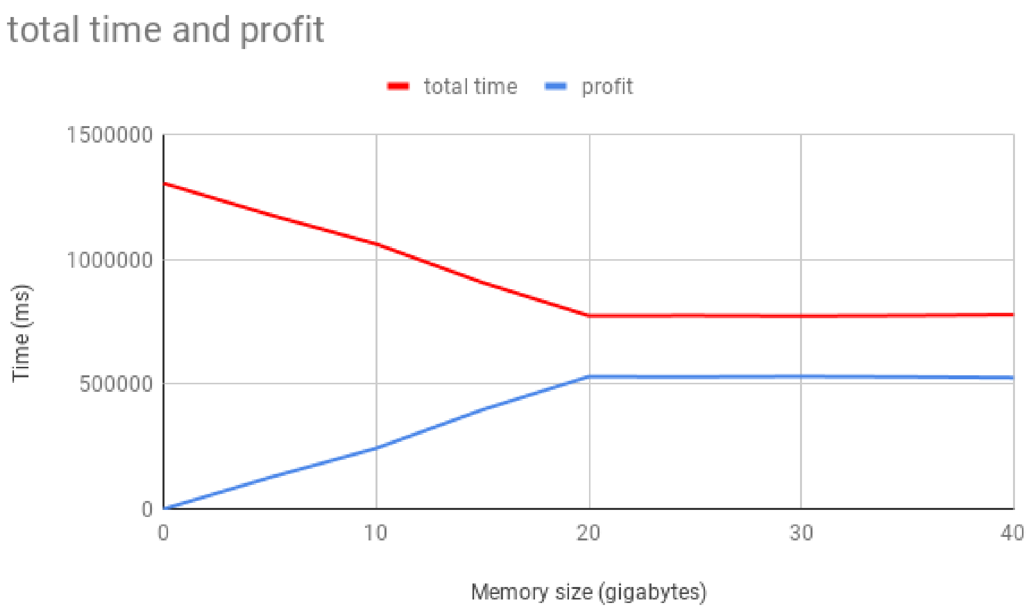

FIGURE 5.2: The total execution time of a query workload with a split in training- and testdata.

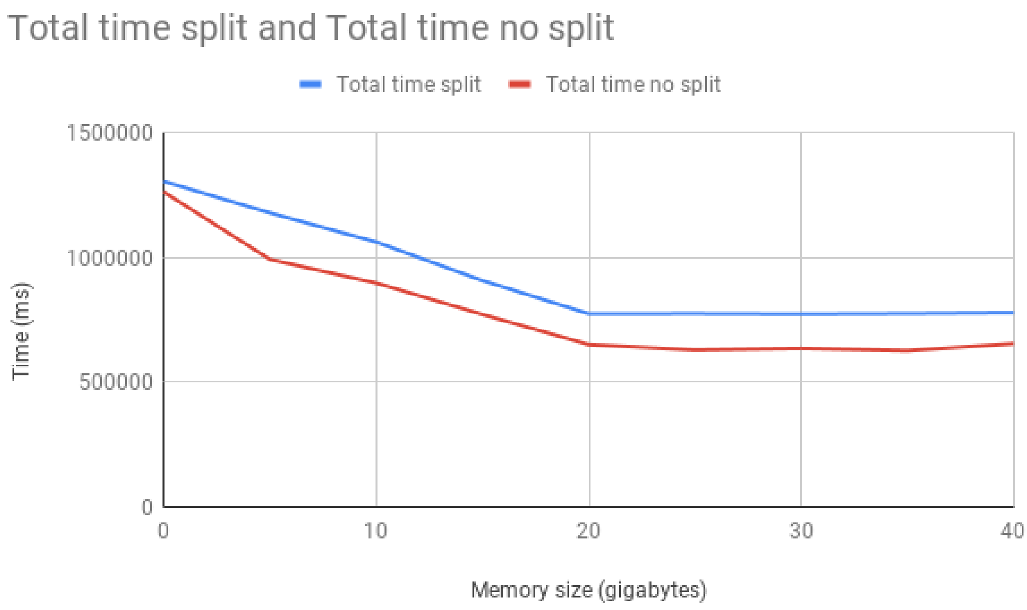

FIGURE 5.3: The total execution time of a split query workload and a workload without.

To find the amortization period I ran the workload a second time. As we can see from figure 5.6, running the workload 2 times shows that materializing subexpressions and running the workload is nearly identical to just running the workload. This would mean that for this setup, the amortization period is around 800 queries.

Would we increase the workload to a size greater than the amortization period, we can see that the performance starts to increase by quite a lot, and this performance would only increase with increasing amounts of workload sizes 5.7.

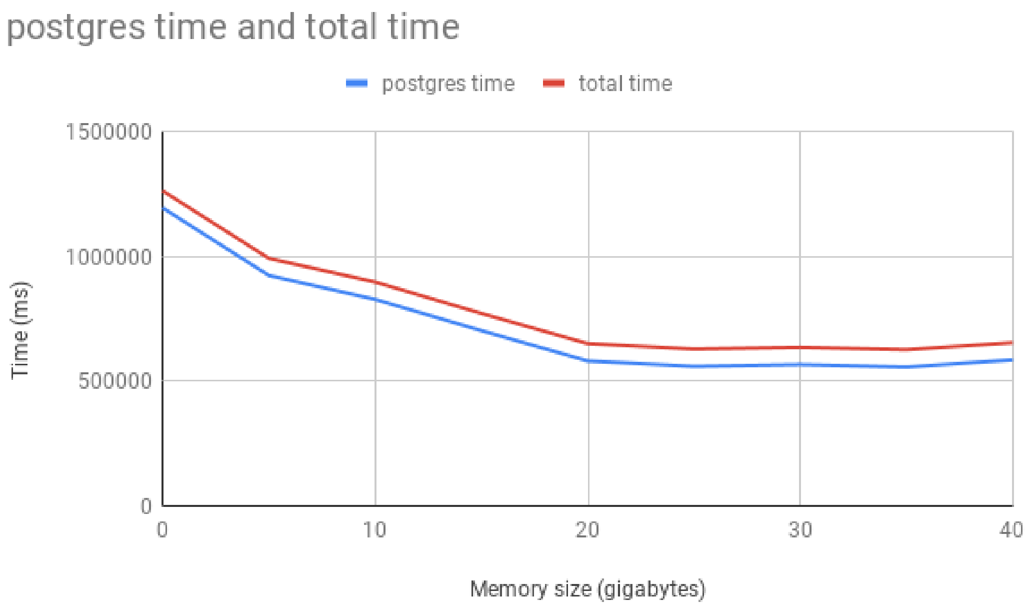

FIGURE 5.4: Execution time of the query workload in Postgres and the total execution time using the same training and test set.

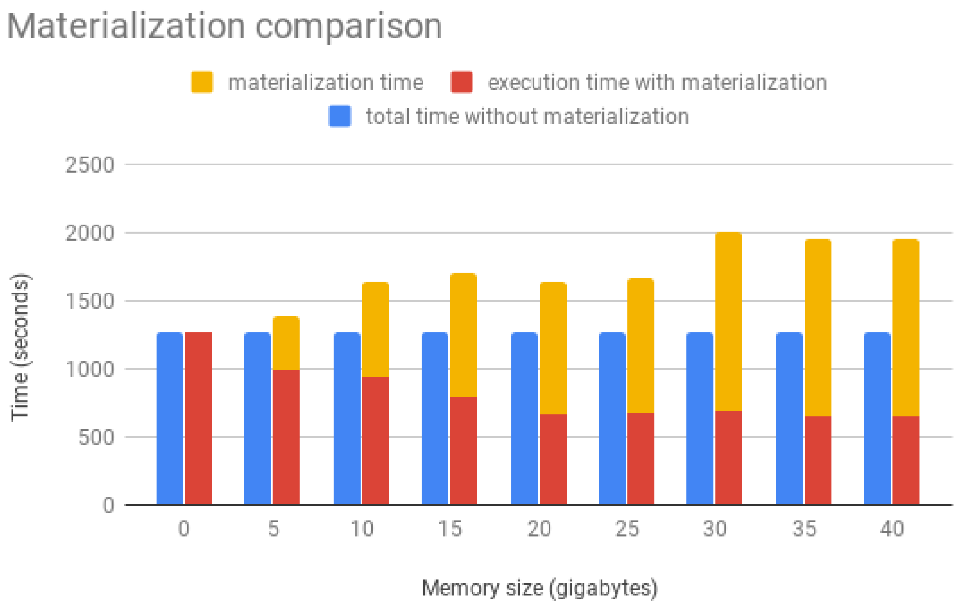

FIGURE 5.5: The total execution with and without materialization.

These results might not be completely accurate since we are comparing an explain analyze command to a materialization command followed by an explain analyze command. Since an explain analyze command does not have to write any results to an outputstream, and materializing a subexpression does, we might get some skewed results with the difference between those. Would this be taken into

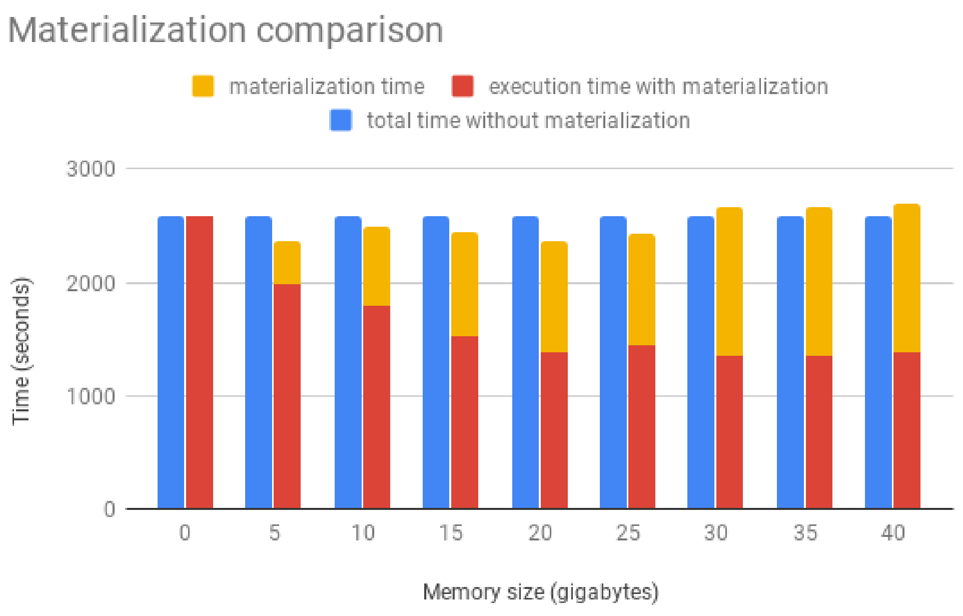

FIGURE 5.6: The total execution with and without materialization with 2 executions of the query workload.

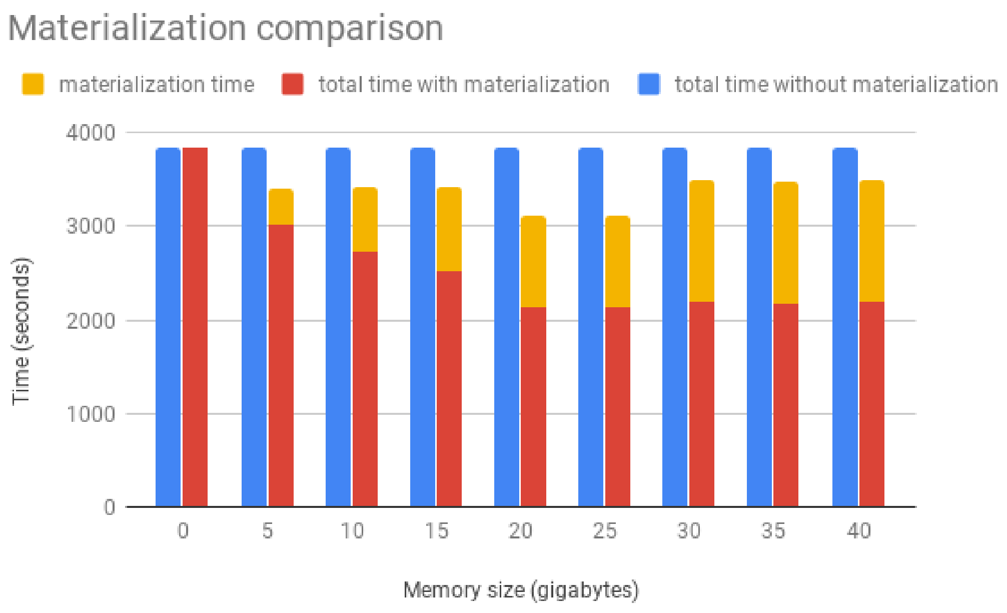

FIGURE 5.7: The total execution with and without materialization with 3 executions of the query workload.

consideration, we should see that materializing subexpressions would have relatively less of an impact with regard to running the query and therefor the amortization period would be lower.

5.4 Analyzing rewrite time

An important aspect of the query rewriting is the time it takes to rewrite these queries. To analyze the impact of the heuristic rewriter of Apache Calcite of the query workload, I have measured the time it takes to rewrite increasing number of queries from 100 to 1600 with jumps of 100 for 4 different amounts of materialized views in the system (0, 20, 40, 80). By measuring the optimization time with 0 materialized views in the system, we can measure the overhead of the rewriting.

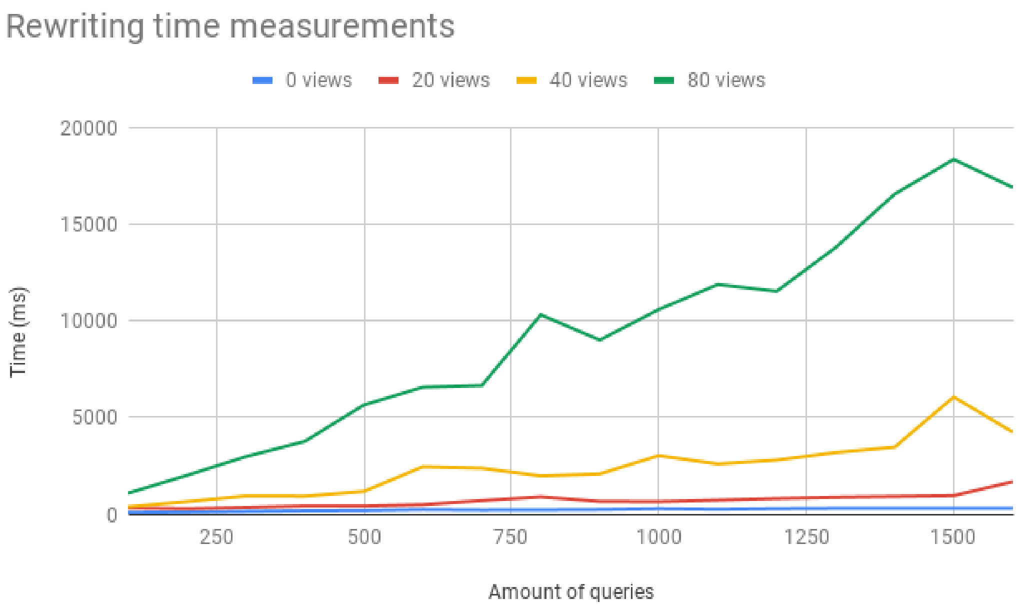

FIGURE 5.8: The time measured to rewrite with increasing amount of queries.

From the measurements in figure 5.8 it can be seen that the time follows a somewhat linear equation with the amount of queries with some overhead (materialized views equals to 0). In other words, for the optimization time there is a linear model following provided that the amount of materialized views is constant:

$$aQ + O = time$$

Where Q is the amount of queries and O is the overhead of the optimizer and a is the linear coefficient dependent on the amount of materialized views. As x increases, the relative effect of the overhead decreases. In further analysis I leave the overhead out of the analysis since its impact is minimal with a large quantity of queries and materialized views

It can also be noted that increasing the amount of materialized views also increases the rewriting time.. To visualize the role of the amount of materialized views I have plotted the time it takes to rewrite 4 sets of queries against increasing amounts of materialized views. These results can be found in figure 5.9

From this figure we can conclude that the amount of materialized has a quadratic impact on the time it takes to rewrite these queries. Updating the formula, we then find that

$$aV^2Q = time$$

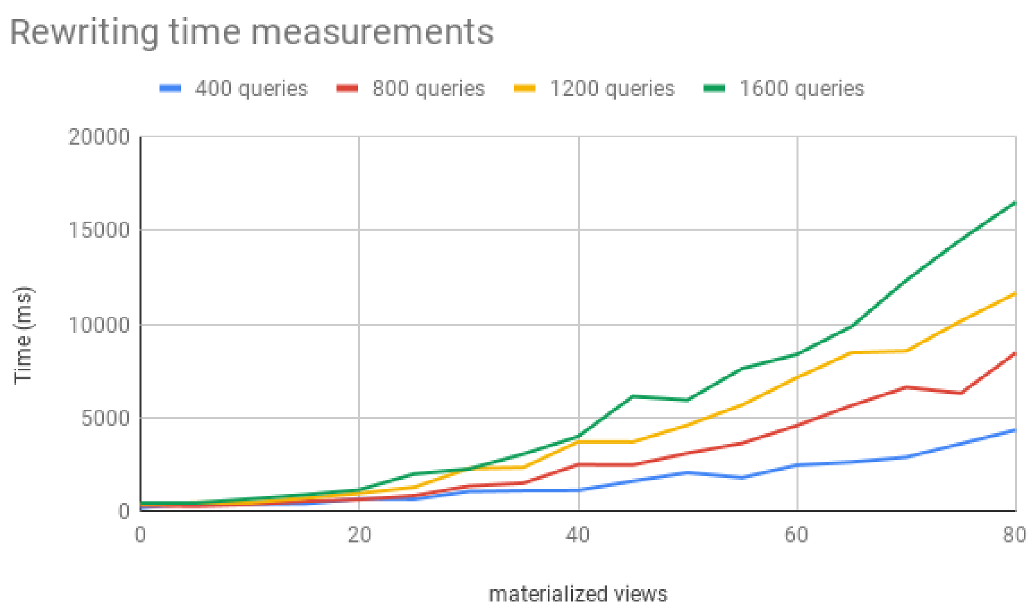

FIGURE 5.9: The time measured to rewrite increasing amount of materialized views.

Where V is the amount of materialized views and Q is the amount of queries.

Finally, to find the linear coefficient we divide both sides by $(V^2 * Q)$. This gives the following scatters of sample coefficients

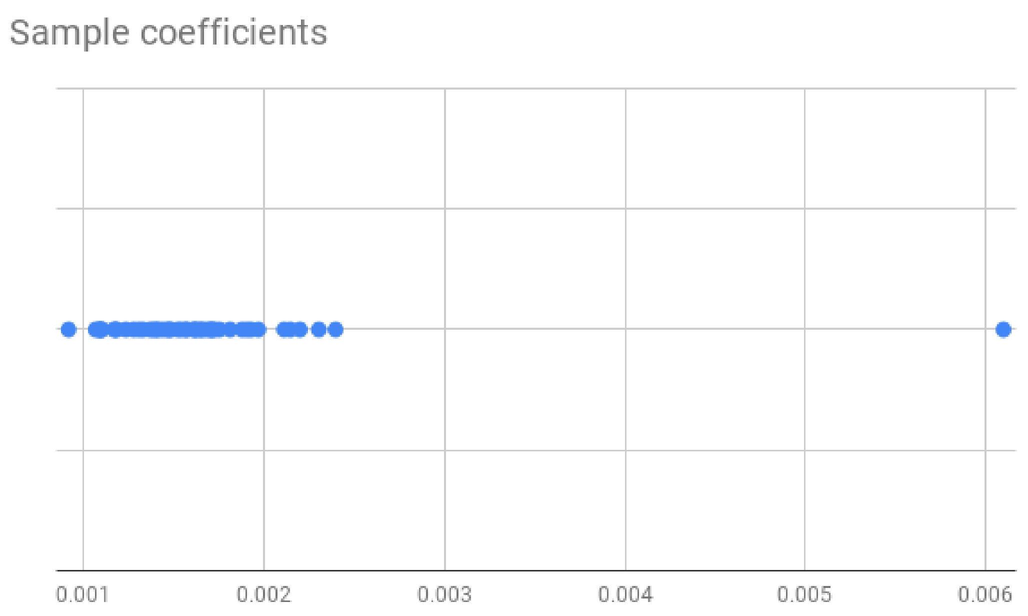

FIGURE 5.10: The sample coefficients for estimating the rewriting time.

Keeping in mind that due to various reasons, these measurements are not perfect, and the sample linear coefficients l will all have various amount margins of error according to:

$$l = a + e$$

Where l is the sample linear coefficient, a is the linear coefficient and e is the error. By taking the median of these coefficients I can reduce the overall error to a minimum. This makes the final equation equal to:

$$time = 0.00155 * V^2 * Q$$

Where $time$ is the amount of time in milliseconds, V the amount of materialized views and Q the amount of queries.

5.5 Summary

In this chapter we have taken a look at the evaluation of the results. We started off with a general experiment setup in section 5.1. Then, in section 5.2 we have taken a look at the results found promising results with around 50% increase in performance using the same training set and test set and around 35% increase in performance using different training sets and test sets.

In section 5.3, we discussed the concept of the amortization period: the amount of queries it would take to start to see an increase in performance. In our experiment, with workload sizes of 400, we found that we had an amortization period of around two workloads which equates to 800 queries.

Finally, we have also taken a more in-depth look at the rewriting time of the rewriter in section 5.4. From the results we found that the runtime of the rewriting algorithm scales linearly with the amount of queries and quadratically with the amount of registered materialized views. Using linear regression, we found that on our system the runetime roughly equates to 0.00155*V^2 * Q, where V is the amount of registered materialized views and Q the amount of queries.

Chapter 6

Threats to validity

In this chapter we discuss the threats that could possible influence the validity of the thesis. For each threat we will go into detail on why it is a threat, the impact it can have and what can be done about it in future work.

6.1 Dataset

One of the main threats to validity towards this thesis is the dataset used in the experiments. Finding a decently sized and representative query workload with its respective data was not possible due to various reason with one of which being privacy concerns. This led to the decision to use simple generated queries instead with the risk being that this these generated queries do not represent a more realistic query workload.

Although I do not suspect the influence on the results to be an issue, there is no way to know this for certain unless we test it on a real query dataset. Resolving this threat to validity would thus imply testing the thesis to a real query workload and not a generated one.

6.2 Subexpression extraction

When creating subexpressions from the different queries in the query workload we chose to define a subexpressions as each subtree in the query operator tree. However, this set of subexpressions does not include all possible subexpressions in a query. If other possible rewritings exists for the query operator tree then that would result in an increased amount of subexpressions. For this thesis I opted to no look into alternative rewritings since it would add a lot of complication to the overall process, but it could be something to incorporate in the future.

Extracting more subexpressions from the queries than we did during the thesis would solely increase the search space, and I would expect the results to be at least as good as what we currently have.

6.3 Evaluation

For the evaluation of the queries we used an explain analyze command to just compute the query and avoid any IO-cost. This is not an accurate representation of query runtime and simple tests on our machine indicated that actually running the query takes longer than the indication by an explain analyze command.

I would not expect the use of explain analyze to have too much of an impact. The advantage of using explain analyze is that we get a fairer representation of the runtime than we would get running the query. Furthermore, I would expect the extra IO-cost of writing the query results would be somewhat equal for the queries with regard to their result set size.

We could eliminate this threat by actually writing out the results for these queries. For obvious reasons I opted not to since the machine I am running the experiments on is not made for such high IO-operations.

6.4 Summary

In this chapter we discussed the potential threats to the validity of the thesis. We have highlighted the three main threaths to the validity: the lack of proper dataset in 6.1, the method of extracting the subexpressions in 6.2 as well as how we evaluated runtime results in 6.3. For these threats we discussed why they are a threat to the validity, as well as the potential impact they can have and what we could have done to avoid them.

Chapter 7

Future work

From chapter 6 threats to validity, we can already find some shortcomings with the current approach, which can be improved in the future. Most importantly of which is finding a decently sized representative query workload.

Other angles that could be explored in future work could be with regard to the approach taken in general. In this thesis we opted to apply a simple and fast heuristic approach since we wanted to have a decent solution as quickly as possible. Perhaps other approaches could achieve better solutions while having similar performance costs. As long as the time gained having a better solution against the extra time it takes of finding that better solution is a net win with regard to the overall time cost it would be beneficial to find such approaches.

7.1 Query sequences

Another topic of future research could be to identify whether queries could be correlated with each other. During the thesis we look at queries as individual units. But if instead we look at queries as units in a sequence, we could detect whether a single query could, an indicator query, could tell us what next queries could occur in the sequence. In contrast to this thesis, where we used queries in the past to optimize past queries assuming that it will be a representation for future queries. Instead, by predicting future queries we can use past queries to train a model to preoptimize queries, which have not been submitted yet.

7.1.1 Collaborative filtering

One way to achieve this is via collaborative filtering (Herlocker, Konstan, and Riedl, 2000). In short, collaborative filtering is a technique that tries to recommend new things one might like using historical user data as well as the data of people that behave similar to the way you do. These systems have been studied extensively and have become very successful in a web context (Adomavicius and Kwon, 2007, Bell, Koren, and Volinsky, 2007, Park and Pennock, 2007).

As an example, say I am really into horror movies, and I would like to know whether I would like a particular horror movie. Collaborative filtering systems would then match me to similar people who have watched that particular movie and whose ratings correspond to my rated movies. Based on their rating of that particular movie the collaborative filtering system would then give me a predictive score whether I would like a certain movie or not. That these systems are important can be seen from the *Netflix prize* contest.

Typically, these systems are used in recommender systems. Research in its application in query sequence optimization has been limited (Akbarnejad et al., 2010, Chatzopoulou, Eirinaki, and Polyzotis, 2009, Marcel and Negre, 2011, Giacometti et al., 2009). Would we be able to predict future queries we can already precompute certain queries to increase the future query performance. This can be extremely helpful in applications where successive queries from users are related to each other like exploratory analysis tasks (Seltman, 2018).

7.2 Summary

In this chapter we have looked at the potential future work of the thesis. We first started with the potential threats to validity from chapter 6. Then we argued that finding better and faster solutions could be something to look into but we also argued that this might not necessarily be the most rewarding way forward.

Instead, in section 7.1 we identified a more interesting gap in research. We discussed that queries have mostly been seen as individual units but there has not been much literature into reasoning about them as related sequences and how we could predict and preoptimize future queries. In section 7.1.1 we then argued how we could for instance use collaborative filtering techniques to accomplish this task as well as providing some preliminary literature into this topic.

Chapter 8

Conclusion

During the thesis we have looked at the effectiveness of materializing query subexpressions and rewriting the existing queries to use said subexpressions. We saw a significant increase in performance with around 50% reduction in execution time in the most optimal condition (train and test set consists of the same data) and around 35% reduction with the train and test set split. Then in the future work we also discussed another interesting approach where,instead of optimizing past queries using historical query data, we could try to predict future queries by training a prediction model and try to eliminate common subexpressions in those future queries to optimize those queries before they are even queried.

This chapter marks the final conclusion of the thesis. In this chapter we will revisit the research question posed in chapter 1.

8.1 Conclusion

Over the thesis we have looked at 4 main research questions related to optimizing query performance by materializing subexpressions.

RQ1: How can we implement the existing state of the art of finding and materialing common subexpressions among queries in an query workload in 2019 using open source software?

Using Apache Calcite and Postgres accomplished much of what we wanted. There were a few difficulties along the way, most of which due to the way of how Apache calcite can be difficult to implement 4.4. In the ideal situation we would have Apache Calcite send the queries to Postgres using a JDBC-connection. Instead, we relied on textual representations of queries and using a management process to first rewrite it using Apache Calcite and then send the rewritten query to Postgres.

Overall the final results are quite promising. With around a 35% reduction in query performance in a more accurate use case 5.

RQ2: How can we efficiently extract candidate subexpressions from the query workload?

The current approach of extracting the query subexpression does not take into consideration any alternative ways of writing the query operator tree. The approach taken, extracting subexpressions as subtrees from the query operator tree, works extremely well and fast. But to make it recognize alternative subexpressions some form of tree rewriting needs to be implemented to not miss those subexpressions.

RQ3: How can we efficiently evaluate candidate subexpressions to find a good solution?

For the evaluation of the subexpressions we wanted something good and fast sometimes sacrificing better solutions in favor of starting the process of executing the queries sooner. The goal of the thesis is to improve query execution performance by materializing common subexpressions. Would we spend more time on finding better solutions then that better solution has to make up for the time difference of finding that solution.

Another goal set out for the evaluation of the subexpressions is that it should be a general algorithm. We wanted to avoid creating an algorithm that could work extremely well in different and niche scenarios but fails to find reasonable solutions in other scenarios. This does not mean however that it will not be beneficial to do so. We could create such an algorithm for such niche scenarios and use our general algorithm as a backup.

Four our algorithm we opted for a custom heuristic solution closely resembling a heuristic knapsack solution. For an approximation solution it worked really well and fast and yielded quite promising results for its fast execution and broad application of the algorithm.

RQ4: How can we rewrite queries to use existing materialized views?

The decision to use an off the shelf rewriter, Apache Calcite in our research, turned out to be a good choice. Not only did we save the time of creating our own rewriter, but we also ended up with a much more sophisticated rewriter. Would we have created our own its functionality would have been limited to single in place rewriting where the accuracy depends on the detection of subexpressions in a query operator tree. By leveraging the rewriter of Apache Calcite we know we have a fully functional and sophisticated rewriter developed over multiple iterations.

The rewriter of Apache Calcite is also quite fast 5.4. Another advantage is that it can also be quite scalable. We found that the rewrite time scales quadratically with the amount of materialized views and linearly with the amount of queries to rewrite. Depending on the scale of the application, we could further increase the rewrite time by making the rewriter part of a distributed system with clusters based on the materialized views, and the queries spread among the machines. A manager process can then distribute the queries based on the table accesses defined in the queries. This would further reduce the rewriting time.

Concludingly, the rewriter Apache Calcite proved to be quite capable in handling the rewriting.

Bibliography

Adomavicius, Gediminas and YoungOk Kwon (2007). “New recommendation techniques for multicriteria rating systems”. In: *IEEE Intelligent Systems* 22.3, pp. 48–55.

Agrawal, Sanjay, Eric Chu, and Vivek Narasayya (2006). “Automatic physical design tuning: workload as a sequence”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, pp. 683–694.

Akbarnejad, Javad et al. (2010). “SQL QueRIE recommendations”. In: *Proceedings of the VLDB Endowment* 3.1-2, pp. 1597–1600.

Bell, Robert, Yehuda Koren, and Chris Volinsky (2007). “Modeling relationships at multiple scales to improve accuracy of large recommender systems”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 95–104.

Chatzopoulou, Gloria, Magdalini Eirinaki, and Neoklis Polyzotis (2009). “Query recommendations for interactive database exploration”. In: *International Conference on Scientific and Statistical Database Management*. Springer, pp. 3–18.

Chirkova, Rada, Alon Y Halevy, and Dan Suciu (2002). “A formal perspective on the view selection problem”. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 11.3, pp. 216–237.

cost.h. `https://github.com/postgres/postgres/blob/8255c7a5/src/include/optimizer/cost.h`. Accessed: 2019-05-29.

costsize.c. `https://github.com/postgres/postgres/blob/d890fa81/src/backend/optimizer/path/costsize.c#L211`. Accessed: 2019-05-29.

Finkelstein, Sheldon (1982). “Common expression analysis in database applications”. In: *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. ACM, pp. 235–245.

Gantz, John and David Reinsel (2012). “The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east”. In: *IDC iView: IDC Analyze the future* 2007.2012, pp. 1–16.

Giacometti, Arnaud et al. (2009). “Query recommendations for OLAP discovery driven analysis”. In: *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*. ACM, pp. 81–88.

Giannikis, Georgios et al. (2014). “Shared workload optimization”. In: *Proceedings of the VLDB Endowment* 7.6, pp. 429–440.

Goldstein, Jonathan and Per-Åke Larson (2001). “Optimizing queries using materialized views: a practical, scalable solution”. In: *ACM SIGMOD Record*. Vol. 30. 2. ACM, pp. 331–342.

Gupta, Himanshu and Inderpal Singh Mumick (2005). “Selection of views to materialize in a data warehouse”. In: *IEEE Transactions on Knowledge and Data Engineering* 17.1, pp. 24–43.

Herlocker, Jonathan L, Joseph A Konstan, and John Riedl (2000). “Explaining collaborative filtering recommendations”. In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM, pp. 241–250.

Jarke, Matthias (1985). “Common subexpression isolation in multiple query optimization”. In: *Query Processing in Database Systems*. Springer, pp. 191–205.
Jindal, Alekh et al. (2018). “Selecting subexpressions to materialize at datacenter scale”. In: *Proceedings of the VLDB Endowment* 11.7, pp. 800–812.
Karanasos, Konstantinos, Asterios Katsifodimos, and Ioana Manolescu (2013). “Delta: Scalable data dissemination under capacity constraints”. In: *Proceedings of the VLDB Endowment* 7.4, pp. 217–228.
Kleinberg, J. and E. Tardos (2013). *Algorithm Design: Pearson New International Edition*. Pearson Education Limited. ISBN: 9781292037042. URL: https://books.google.nl/books?id=ayOpBwAAQBAJ.
Koutroumbas, K. and S. Theodoridis (2008). *Pattern Recognition*. Elsevier Science. ISBN: 9780080949123. URL: https://books.google.nl/books?id=QgD-3Tcj8DkC.
Mack, Chris A (2011). “Fifty years of Moore’s law”. In: *IEEE Transactions on semiconductor manufacturing* 24.2, pp. 202–207.
Mami, Imene and Zohra Bellahsene (2012). “A survey of view selection methods”. In: *ACM SIGMOD Record* 41.1, pp. 20–29.
Marcel, Patrick and Elsa Negre (2011). “A survey of query recommendation techniques for data warehouse exploration.” In: *EDA*, pp. 119–134.
Mistry, Hoshi et al. (2001). “Materialized view selection and maintenance using multi-query optimization”. In: *ACM SIGMOD Record* 30.2, pp. 307–318.
Netflix prize. https://www.netflixprize.com/. Accessed: 2019-10-28.
Park, Seung-Taek and David M Pennock (2007). “Applying collaborative filtering techniques to movie search for better ranking and browsing”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 550–559.
Pisinger, David (2007). “The quadratic knapsack problem—a survey”. In: *Discrete applied mathematics* 155.5, pp. 623–648.
Ross, Kenneth A and Divesh Srivastava (1997). “Fast computation of sparse datacubes”. In: *VLDB*. Vol. 97. Citeseer, pp. 25–29.
Roussopoulos, Nick (1982). “The logical access path schema of a database”. In: *IEEE Transactions on Software Engineering* 6, pp. 563–573.
Salkin, Harvey M and Cornelis A De Kluyver (1975). “The knapsack problem: a survey”. In: *Naval Research Logistics Quarterly* 22.1, pp. 127–144.
Sellis, Timos K (1988). “Multiple-query optimization”. In: *ACM Transactions on Database Systems (TODS)* 13.1, pp. 23–52.
Seltman, Howard J. (2018). *Experimental Design and Analysis*.
Simonite, Tom (2016). “Intel puts the brakes on Moore’s Law”. In: *MIT Technology Rev.*
StackOverflow. https://stackoverflow.com/. Accessed: 2019-11-19.
table not found with apache calcite. https://stackoverflow.com/questions/31118348/table-not-found-with-apache-calcite. Accessed: 2019-09-01.
Tanenbaum, A.S. and M. van Steen (2013). *Distributed Systems: Principles and Paradigms*. Always learning. Pearson Education Limited. ISBN: 9781292025520. URL: https://books.google.nl/books?id=5tA-nwEACAAJ.
TPC-DS. http://www.tpc.org/tpcds/. Accessed: 2019-09-25.
Zhou, Jingren et al. (2007). “Efficient exploitation of similar subexpressions for query processing”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, pp. 533–544.