**The Open University of Israel**

**Department of Mathematics and Computer Science**

**Visualizing Database Execution Plans using Sankey**

שימוש בגרף סאנקי לוויזואליזציה של תכנית הרצה של מסדי נתונים

Thesis proposal submitted as partial fulfillment of the requirements

towards an M.Sc. degree in Computer Science

The Open University of Israel

Computer Science Division

By

**Eyal Trabelsi**

Prepared under the supervision of Prof. Ehud Gudes

2021

# Acknowledgments

I would like to thank my thesis instructor Prof. Gudes from the department of computer science at the open university, for reviewing my work, providing helpful tips, and investing time in our research.

# Abstract

Database management systems (or DBMS) have been around for decades, and thanks to their scalability, and usability characteristics they have become a critical part of every application. Yet, they are still difficult to use, in particular, it’s hard to identify flaws in both users’ queries and the database’s configuration.

Databases are complex systems, and often lack the tooling to identify flaws and their origins. Without proper toolings, such as profilers and debuggers, it is rare for users to write flawless queries.

In our work, we focus on identifying cardinality problems and performance bottlenecks. Our goal is to provide a tool to identify flaws in SQL queries, that work with existing real-world systems. Thus, we assume that our solution shouldn't require change in the databases themselves.

We have implemented QueryFlow, a tool to understand and visualize queries’ characteristics. This helps identify common DBMS’ problems, such as performance bottlenecks and cardinality issues. QueryFlow is extendable to new databases. Onboarding a new database only requires the implementation of an execution plan parser.

The experimental results show that our solution allows us to pinpoint and fix the flaws in real world queries.

# Table of Content

Abstract ……………………………………………………………………3  
Table of content ……………………………………………………........... 4   
Figure list ……………………………………………………………......... 5  
Chapter 1: Introduction …………………………………………………… 9  
Section 1.1: Problem description …………………………………….......... 9  
Section 1.2: Motivation ……………………………………………............. 9  
Section 1.3: Main Contributions…... ………………………………............. 9  
Section 1.4: Thesis Structure ……………………………………………… 10  
Chapter 2: Background and Related work ………………………………... 11  
Section 2.1 Common Flaws ………………………………………….......... 12   
Section 2.2 Execution Plan Overview ……………………………………... 13  
Section 2.3: Static Analysis ………………………………………….......... 14  
Section 2.3.1: Static Analysis Introduction ………………………….......... 14  
Section 2.3.2: The Empty Answer Problem…………………………….......... 14  
Section 2.3.3: The Why and Why not Problems ………………………….. 14  
Section 2.4: Data Governance………………………………………… 16  
Section 2.5: Debugging Approach ………………………………………… 16  
Section 2.6: Query Visualization Approach …………………………... …..17  
Section 2.6.1: Query Visualization Introduction…………………………... …..17  
Section 2.6.2: Visualize the Logical Execution Plan …………………………... …..17  
Section 2.6.3: Visualize the Actual Execution Statistics …………………………... …..17  
Section 2.6.4: Visualize the Optimizer Statistics …………………………... …..17  
Section 2.7: Approaches Comparison …………………………………….. 20  
Section 2.8: Multi Query Optimization …………………………………… 21  
Section 2.9: Sankey Diagrams …………………………………………….. 22  
Section 2.10: Summary ……………………………………………………... 24  
Chapter 3: QueryFlow Design …………………………………………….. 26  
Section 3.1: QueryFlow design ………………………………………......... 27  
Section 3.2: QueryFlow Parsing …………………………………………… 28  
Section 3.3: QueryFlow Enrichment ………………………………………. 29  
Section 3.4: QueryFlow Visualization …………………………………. … 30  
Section 3.5: QueryFlow Detailed Example ……………………………...….. 32  
Section 3.5.1: Example Definition……………………………...….. 32  
Section 3.5.2: Getting the Database Execution Plan ……………………………...….. 32  
Section 3.5.3: QueryFlow’s Parsing Phase……………………………...….. 32  
Section 3.5.4: QueryFlow’s Enrichment Phase……………………………...….. 32  
Section 3.5.5: QueryFlow’s Visualization Phase……………………………...….. 32  
Chapter 4: QueryFlow Use-cases ………………………………………….... 44  
Section 4.1: Identifying missing records …………………………………..… 45  
Section 4.2: Identifying Ineffective Operations ………………………..……. 47  
Section 4.3: Identifying Duplications …………………………………..…… 49  
Section 4.4: Identifying Performance Bottlenecks in a Single Query …..…... 52  
Section 4.5: Identify flaws in the optimizer itself ……….………………….. 58

Section 4.6: Identifying Performance Bottlenecks in Multiple Queries .…… 54  
Section 4.7: When QueryFlow won’t help …………….…………………… 60  
Chapter 5: Evaluation ………………………………..…………………..…. 63  
Section 5.1: TPC-H ………………………………….…………………..…. 64  
Section 5.1: Evaluation setup ……………………….……………………… 65  
Section 5.2: Evaluate optimizations for scale factor 1 ……………………... 66  
Section 5.3: Evaluate optimizations for scale factor 10 ……………………. 68  
Conclusions and Future Directions ……………………………………....... 70  
References …………………………………………………………………. 71

# Figure list

Figure 1: Taxes table before modification (taken from QFix paper) ……………………………….………….. 15

Figure 2: Modifications for taxes table (taken from QFix paper) …………………………………………..….. 16

Figure 3: Taxes table after modification (taken from QFix paper) ………………………………………. …….17

Figure 4: query's breakpoint interface (Taken from Habitat paper) ……………………………………….…… 17

Figure 5: query sub-expression materialized view (Taken from Habitat paper) ………………………………. 18

Figure 6: complex query’s SQL representation (Taken from QueryViz paper) ……………………………. … 19

Figure 7: complex query’s ERD-like representation (Taken from QueryViz paper) ………………….……….. 19

Figure 8: Sankey Diagram of Sales Funnel of both teams ……………………………………………….…….. 22

Figure 9: Additional information pop up when hovering an edge ……………………………………….…….. 23

Figure 10: Sankey diagram of the energy ecosystem (Taken from paper 18) ……………………..................... 23

Figure 11: QueryFlow Design ……………….…………………………………………………………….. . …28

Figure 12: IMDB Schema ……………………………………………………………..………………….…….32

Figure 13: Detailed Example – Query ……………………………………………..………………………........33

Figure 14: Detailed Example - Explain Query ……………………………………………………………….....33

Figure 15: Detailed Example - Explain Analyze Query …………………………………………….……….…34

Figure 16: Detailed Example - Explain Analyze Results ……………………………………………………….34

Figure 17: Detailed Example - Explain Results …………………………………………………………………35

Figure 18: Detailed Example - Tree representation of execution plan …………………………….……………36

Figure 19: Detailed Example – QueryFlow’s Parsing algorithm …………………………….………………….36

Figure 20: Detailed Example - Tree representation of execution plan after QueryFlow parsing phase ……...…37

Figure 21: QueryFlow’s Enrichment algorithm…………………………………………….………………… ...30

Figure 22: Detailed Example - Tree representation of execution plan after QueryFlow enrichment phase …….40

Figure 23: Detailed Example - Table representation of execution plan after QueryFlow enrichment phase …...41

Figure 24: QueryFlow’s Visualization algorithm…………………………………………….………………… 30

Figure 25: Detailed Example - Table representation of execution plan for QueryFlow visualization phase …...41

Figure 26: Detailed Example - QueryFlow Sankey for sub-expression cardinality ………………………….…42

Figure 27: Detailed Example - QueryFlow Sankey for sub-expression duration ………………………………42

Figure 28: Identify Missing Records – Query …………………………………………………………………...45

Figure 29: Identify Missing Records - QueryFlow Sankey for query’s cardinality …………………………….45

Figure 30: Identify Missing Records - Fixed Query ………………………………………………………..…..46

Figure 31: Identify Missing Records - QueryFlow Sankey for fixed query's cardinality …………………...…46

Figure 32: Identify Ineffective Operations – Query ………...……………………………………………….…47

Figure 33: Identify Ineffective Operations - QueryFlow Sankey for query's cardinality …………………….. 48

Figure 34: Identify Ineffective Operations - Fixed Query ……………………………………………………. 48

Figure 35: Identify Ineffective Operations - QueryFlow Sankey for fixed query's cardinality ………………..49

Figure 36: Identify Duplications – Query …………………………………………………………………….. 50

Figure 37: Identify Duplications - QueryFlow Sankey for query's cardinality …….. …………………………50

Figure 38: Identify Duplications - Fixed Query ……………………………………………………………… 51

Figure 39: Identify Duplications - QueryFlow Sankey for fixed query's cardinality. …………………………51

Figure 40: Identify Performance Bottleneck in a Single Query – Query ……………………………...………...52

Figure 41: Identify Performance Bottleneck in a Single Query -   
 QueryFlow Sankey for query's cardinality………………………………………….……………….. 52

Figure 42: Identify Performance Bottleneck in a Single Query - QueryFlow Sankey for query's duration……. 53

Figure 43: Identify Performance Bottleneck in a Single Query - Index creation………………………………. 53

Figure 44: Identify Performance Bottleneck in a Single Query -   
 QueryFlow Sankey for query's duration after optimization ………………………………………… 54

Figure 45: Identify Performance Bottleneck in Multiple Queries -   
 Modifications to the visualization algorithm for multiple metrics visualization.……………………54

Figure 46: Identify Flaws in the Optimizer – Query ……………………………………………………………58

Figure 47: Identify Flaws in the Optimizer -   
 QueryFlow Sankey for query's cardinality and estimated cardinality………………….……………59

Figure 48: Identify Flaws in the Optimizer – Vacuum …………………………………………………………59

Figure 49: Identify Flaws in the Optimizer -   
 QueryFlow Sankey diagram for query's cardinality and estimated cardinality after vacuum………60

Figure 50: Identify Performance Bottleneck in Multiple Queries -  
 Modifications to the parsing algorithm for multiple queries ……….………………………………58

Figure 51: Identify Performance Bottleneck in Multiple Queries -  
 Modifications to the visualization algorithm for multiple queries visualizations………………..…58

Figure 52: Identify Performance Bottleneck in Multiple Queries - Query 1 …………………………….……54

Figure 53: Identify Performance Bottleneck in Multiple Queries - Query 2 …………………………….……55

Figure 54: Identify Performance Bottleneck in Multiple Queries -   
 QueryFlow Sankey for queries' cardinality ………………………………………………………..55

Figure 55: Identify Performance Bottleneck in Multiple Queries -   
 QueryFlow Sankey for queries' duration ………………………………………………………….56

Figure 56: Identify Performance Bottleneck in Multiple Queries - Crew Index creation ………..………….56

Figure 57: Identify Performance Bottleneck in Multiple Queries -   
 QueryFlow Sankey for queries' duration with Crew index……………………………………….57

Figure 58: Identify Performance Bottleneck in Multiple Queries - Titles Index creation …………………57

Figure 59: Identify Performance Bottleneck in Multiple Queries -   
 QueryFlow Sankey for queries’ duration with Title index ………………………………………...58

Figure 60: When QueryFlow won’t help - Skew problems ……………………………………………………..60

Figure 61: When QueryFlow won’t help - Skew problems mitigation ………………………………………….61

Figure 62: When QueryFlow won’t help - complexity problem ………………………………………………61

Figure 63: TPC-H Schema ………………………………………………………………………………………64

Figure 64: TPC-H SF-1 Baseline Histogram...………… ………………………………………...64

Figure 65: TPC-H SF-1 Sankey-diagram for all queries …………………………………………………..…… 65

Figure 66: TPC-H SF-1 Sankey-diagram for slowest queries ………………………………………………….. 65

Figure 67: TPC-H SF-1 Optimized Histogram …………….. …………………………………………….. 66

Figure 68: TPC-H SF-1 Optimized vs Baseline ……………..…………………………………………….. 66

Figure 69: TPC-H SF-1 Sankey-diagram for slowest queries ………………………………………………….. 67

Figure 70: TPC-H SF-10 Sankey-diagram for slowest queries ………………………………………………… 68

# Chapter 1: Introduction

##### 1.1 Problem description

Structure Query Language (SQL) is used for interaction between DBMS and its users. SQL is a declarative query language, designed for managing and manipulating data, and for decades SQL has been the standard for specifying queries over DBMS.  
  
Unfortunately, non-trivial queries are hard to perfect, even for SQL experts. Identifying flaws in complex queries and bringing those queries to perfection is challenging. As SQL queries may return zero entries, duplicate entries, unexpected results, or not meet the minimal performance requirements. Hence, the ability to understand and “debug“ the execution of a SQL query is a necessary step towards using DBMSs effectively.

To mitigate it, most databases’ optimizers will provide the execution plan. When we execute a SQL query, DBMS first parse and validate the query. Then, if the query is valid the optimizer translates the query to a tree representation, called an execution plan. The execution plan is a sequence of steps used to access data to provide the users the result for their query. Understanding the behavior of each step is critical to understanding the query behavior in general.

As DBMS users, we are expected to use the execution plans, to make sense of our queries and to understand our queries’ characteristics. Unfortunately, the execution plans for complex queries can be overwhelming and hard to read. This makes identify flaws in a SQL query a difficult and unintuitive task with the current tooling.

##### 1.2 Motivation

Our aim is to help users find and fix their queries in a more intuitive way. One of the most prominent techniques is to visualize the execution plan. This gives users a more intuitive understanding of how the query is executed (or planned) and a better understanding of the query sub-expressions characteristics.

The goal of this thesis is to give a better way to identify the most common flaws in SQL queries. This is done by parsing the queries’ execution plans and representing them in an intuitive way using Sankey-diagrams. Sankey-diagram [1] is a visualization technique to display any kind of measurable flow. Sankey-diagram is a graph representation with specific characteristics on the nodes and edges. The nodes represent the entity, and the edges represent a measurable metric with proportional width.

##### 1.3 Main Contributions

1. **Execution Plans Parsing –** Databases create execution plans per query, which limits the user’s ability to find and fix flaws like performance bottlenecks across several queries. Also, the sub-expression granularity of the execution plan and the relational operations are different. We mitigate these issues by adding another parsing phase after the database created the execution plan.
2. **Execution Plans Enrichment** – Execution plans include a lot of statistics for our queries. Unfortunately, some useful statistics are missing from the execution plan or may have an unintuitive representation and granularity. We mitigate these issues by:

* Inferring sub-expressions from existing ones. For example, whether a sub-expression is redundant or not.
* Adding sub-expressions statistics with a more intuitive granularity (not cumulative).
* Adding sub-expressions statistics with a more intuitive representation (like percentages).

1. **Execution Plans Visualization** - A new representation for queries as a Sankey diagram. This allows us to understand the nature of a query, several queries, or even the optimizer itself. It can be used to find cardinality issues, bottlenecks, and optimizer problems caused by database configuration.
2. **Examples of Flaws in SQL**- I cover some of the most common flaws in SQL queries. Then I show how to identify those with QueryFlow and how to fix those issues.
3. **Analysis of TPC-H benchmark-** I provide an analysis on the TPC-H benchmark and how the database and the queries can be optimized.

##### 1.4 Thesis Structure

The rest of the thesis is structured as follows:

* Section 2 provides background information and overviews of related work.
* Section 3 presents QueryFlow design for identifying flaws in queries using Sankey-diagrams.
* Section 4 provides an overview of QueryFlow use cases.
* Section 5 provides an evaluation of QueryFlow on the TPC-H benchmark.

Chapter 2: Background and Related work  
As mentioned earlier, the main goal of this thesis is to propose a novel method that allows to identify and fix common flaws, like performance bottlenecks and cardinality issues. In this chapter, we cover relevant background and related work.   
  
The rest of this chapter is structured as follows:

* Section 2.1 provides an overview of the most common problems in SQL queries.
* Section 2.2 provides an overview of the execution plans.
* Section 2.3 covers related work on static analysis of execution plans.
* Section 2.4 covers related work on data governance.
* Section 2.5 covers related work on the debugging approach to identify flaws in SQL queries.
* Section 2.6 covers related work on SQL queries visualization approach to identify errors in SQL queries.
* Section 2.7 provides a comparison between the different approaches to identify errors in SQL queries.
* Section 2.8 covers related work on multiple queries’ optimizations.
* Section 2.9 provides an overview of the Sankey Diagram and its merits.
* Section 2.10 provides a summary.

2.1 Common Flaws

We already mentioned that despite their maturity and popularity, DBMSs are still difficult to use. Improving the usability of database systems is considered an important research topic [2] , particularly when trying to identify and fix flaws. These flaws can originate from the data itself, the query, or the DBMS configuration.  
  
In a data-driven world, where important decisions are based on the data retrieved from DBMS, it is critical to present the information to the stakeholders in an accurate and timely manner. Undetected flaws can cause (but not limited to) the following:

1. Wrong decision-making or missed business opportunities.
2. Increased DBMS cost due to increase in hardware. Since DBMS has multiple queries and each will take longer, we will need more computation hardware to answer all the queries.
3. Bad experience of DBMS’s internal users and waste of their time. For example, the analyst job is iterative, and if the query execution takes a while each iteration is becoming longer.
4. Bad customers’ experience due to slow applications. For example, ideally, a website should take 1 second to load, users may stop using the website if it’s not responsive enough.

The first group of flaws corresponded to an unexpected query’s results. This group consists of but limited to the following flaws :

1. The data is not as expected.
2. Query result with zero entries.
3. Filters that remove no entries.
4. Join between relations that return fewer entries than expected.
5. Join between relations that return more entries than expected.
6. Join between relations that return duplicate entries.
7. The behavior of *Null* operation.

The second family of flaws corresponded to queries’ performance. This group consists but limited to the following flaws:

* Missing indices that cause unoptimized relational operations.
* Unused indices that cause slower updates.
* Missing partitions that cause unoptimized relational operations.
* Unoptimized database configurations that cause a spill to disk.
* Stale statistics can affect the optimizer to pick a sub-optimal execution plan.
* Operations that don’t change the result.
* Operations that help several queries’ performance, but harm others.

Helping DBMS using to identify and fix those issues has been a research topic for a decade and several approaches had been taken:

* Static analysis of the query characteristics, I cover this in detail in chapter 2.3.
* Data governance, I cover this in detail in chapter 2.4.
* Adding debugging capabilities using views, I cover this in detail in chapter 2.5.
* Visualization of the query, I cover this in detail in chapter 2.6.

##### Most of the approaches taken are using the execution plan behind the scenes, to understand the way they work, in section 2.2 I will provide an overview of execution plans.

##### 2.2 Execution Plan Overview

When we execute a SQL query, the DBMS first parse and validate the query. Then, if the query is valid, it translates it to a tree representation, called an execution plan. The execution plan is a sequence of steps used to access data in a DBMS and to provide the users the result for their query.

Due to SQL’s declarative nature, several plans can be derived from a single SQL statement [3], each using different strategies and algorithms. While these different plans will give the same result, the query’s cost will vary depending on the strategies and algorithms executed. The optimizer’s goal is to pick the optimal execution plan, as choosing the best plan will reduce the query’s cost. Unfortunately, finding the best plan is not trivial, in fact [4] claims this problem is NP-complete.

As DBMS users, we are expected to use the execution plan, to understand our queries’ characteristics. Each step of the execution plan corresponds to a relational sub-expression and understanding the behavior of each sub-expression is critical to understanding the query behavior in general.

In general, there are two types of execution plans:

* The logical execution plan - an estimation of how the query will be executed. This will be faster as the query will not be executed, but it will have estimated statistics only.
* The actual execution statistics – the query is being executed. This will be slower as the query will be executed, but it will have both estimated and actual statistics.

Each sub-expression of the execution plan includes the following information:

* The node type represents whether it’s a table scan, a join, an aggregation, or other.
* Information related to the node type, for example, *Filter* for *SCAN* nodes.
* Estimated statistics, like the cost and the *Plan Rows.*
* Actual statistics (only for actual execution statistics), like the time and the *Actual Rows.*

PostgreSQL provides the *EXPLAIN* command, a customizable command to work with execution plans:

* The user can pick *TEXT*, *XML*, *JSON*, or *YAML* as the output format.
* The user can get both the logical execution plan and the actual execution statistics. By default, *EXPLAIN*  return the logical execution plan by, and by adding *ANALYZE* it will return the actual execution statistics.
* The user can add additional statistics using *BUFFERS, COST* and more.

Since the execution plans are flexible and include useful information at the sub-expression level, most of the related work that I cover in section 2.3, section 2.5, and section 2.6 will utilize either the actual execution statistics or the logical execution plan to identify and fix the flaws in our queries.`

##### 2.3 Static Analysis

2.3.1 Introduction  
As we saw in section 2.2, both types of execution plans include valuable information. In this section, we review the relevant literature about using static analysis to understand query behavior and to find flaws.   
  
Although there are hundreds of papers that utilize static analysis to identify flaws in SQL query, most of them are extremely similar and only differ on the heuristics taken. The papers solve the following research problems:

* “The empty answer problem” - This problem occurs when a user writes a restricted query that eliminates all the results. A detailed explanation can be seen in section 2.3.2
* “The why and why not problem” - This technique can help understand why a specific record appears in the result or why it doesn’t appear in the results. A detailed explanation can be seen in section 2.3.3.

2.3.2 The Empty Answer Problem

The papers that tackle the “empty answer problem” focus on generating a less restrictive version of a query. The modified query will return a non-empty result to the user and should be as close to the original query intent as possible.

IQR [5] uses a probabilistic framework to rewrite the query to solve the empty answer problems. To retrieve more results, it needs to find the relational operations where records are being filters and generates potential candidates for relaxation. For example, if we remove a *WHERE* clause or part of it, we might get a non-empty result, these modified queries are our candidates and later we will pick the best candidate using the probabilistic framework. The probabilistic framework takes into account the probability of the user to accept a suggested relaxation, as well as other optimization objectives, such as, minimizing the number of user interactions or returning relevant results.   
  
Tools that solve the “empty answer problem” can be useful, but they come with substantial problems that make them less appealing.

* The modified query might not reflect the user’s intention anymore.
* The “empty answer problem” error is only a fraction of the errors that a user encounter.
* Require query execution.

2.3.3 The Why and Why Not Problem  
  
The papers that tackle the “why and why not problem” focus on understanding the cause for results to appears or be missing in a query result. For example, if I have a specific account and I expect it to be part of the query result and it isn’t, these tools will show us where the record was prone.

Ned Explain [6] solves the “why not” problem by iterating the sub-expressions and check whether a specific value exists or not in that sub-expression. When we know in which sub-expression an entry exists or not, we can pinpoint the earliest sub-expression that misses the expected entry.

Tools that solve the “why and why not problem” can be useful, but they come with substantial problems that make them less appealing.

* Not trivial to know you have an issue in advance.
* The “why and why not problem” error is only a fraction of the errors that the user encounter.
* These systems tend to be complex and hard to maintain.
* Require query execution.

##### 2.4 Data Governance

Data governance defines who can take what action upon what data, in which situations, and using what methods. Thus, data governance strategy is fundamental for any organization that works with data. Data governance is a huge research topic has the following responsibilities (and more):

* **Data security:** ensuring privacy, confidentiality and appropriate access.
* **Data quality:** Improve the quality of your data with validation, data cleansing, and data enrichment.
* **Data integration:** from acquiring or extracting the data to transform and deliver it to a fast access layer.
* **Data monitoring:** collecting and categorizing metadata about your data sources and data pipelines in order to increase visibility.

Out of those responsibilities only data monitoring is closely related to SQL debugging. In particular, techniques that focus on the question “what data change affected my query” are useful. These techniques allow us to understand what queries changed the data itself such as our query doesn’t work as expected.

QFix [7] uses the query log to look at past queries in the query log to identify the ones that affected our query. In the paper we are given an example related to tax rate and payments. Tax brackets determine tax rates for different income levels and are often adjusted.  
In Figure 1 we see the tax before the modification.

Table

Description automatically generated  
 **Figure 1-** Taxes table before modification (taken from QFix paper)  
  
In Figure 2 we describe recent changes to the taxes table. First, we update the tax rate for those with income above $87500 and then we insert new record.

Text

Description automatically generated with low confidence  
 **Figure 2-** Modifications for taxes table (taken from QFix paper)

In Figure 3 we see the taxes table after our modifications, we see that *t5* was not updated with the proper logic.  
Table

Description automatically generated  
 **Figure 3-** Taxes table after modification (taken from QFix paper)

Data debugging is useful but since it occurs before query debugging, we can consider data debugging to be orthogonal to identifying flaws in the query itself.  
  
Some of the cons of debugging data using the query-log are:

* Not trivial to know you have an issue in advance.
* We can’t find issues in the query itself.

##### 2.5 Debugging Approach

In this section, we will review the relevant literature of debugging DBMS queries using views. These tools [8][9] allow users to add breakpoints and retrieve the result of the sub-expression using views. This gives the users a much more granular understanding of the query flow.   
  
In Habitat [9] the user marks the desired sub-expression with a breakpoint and materialize the corresponding sub-expression using views. For example, we can mark the inner query with a breakpoint (*s1*) as can be seen in Figure 4, the corresponding view that was generated from this breakpoint can be seen in Figure 5.

Text

Description automatically generated

**Figure 4-** query's breakpoint interface (Taken from Habitat paper)

As we said in Figure 5, we will see the sub-expression from the breakpoint *s1* materialized, we can see on the left the *p\_partkkey* from *part* relation and the corresponding *ps\_supplycost* from *partsupp* relation. In this example, we can see that *p\_partkkey=3* has two suppliers and if we want to keep only the minimal one, we need to add additional aggregation later.

Table

Description automatically generated with medium confidence

**Figure 5-** query sub-expression materialized view (Taken from Habitat paper)  
  
Debugging SQL queries using views, can be extremely useful in certain situations, but it comes with the following cons:

* Very slow with several breakpoints, as it materialized results for each breakpoint.
* Not trivial to find where the issue was originated from for complex queries.
* These systems tend to be complex and hard to maintain and must be part of the database itself.
* Require query execution.

##### 2.6 Query Visualization Approach

2.6.1 Introduction  
As we saw in section 2.2, both type of execution plans includes valuable information. In this section, we review the relevant literature about using visualization to understand query behavior and find those flaws.   
  
The query visualization papers can be divided as follows:

* Visualize the logical execution plan to make the query more readable.
* Visualize the actual execution statistics to improve performance.
* Visualize the optimizer statistics.

Each type of the query visualizations has pros and cons and knowing what to visualize and how can be very useful.

2.6.2 Visualize the Logical Execution Plan

The first technique focuses on visualizing the logical execution plan of a query, which gives an intuitive understanding of a query. One of the most prominent advantages of using the logical execution plan is that it does execute the query, which makes it a very scalable technique.   
  
QueryViz [10] creates a succinct representation of a query’s logical plan similar to ERD. In Figure 6 we can see a query across many relations including *Actors* and *Cast* and it’s hard to follow the logic behind it.   
 Text

Description automatically generated

**Figure 6-** complex query’s SQL representation (Taken from QueryViz paper)  
  
In Figure 7 we will see the query from Figure 6 represented in an ERD-like representation. **Diagram

Description automatically generated  
 Figure 7-** complex query’s ERD-like representation (Taken from QueryViz paper)

Although Visualization of the logical execution plan can be useful, it comes with the following cons:

* It can provide only a shallow understanding of a query’s characteristics.
* They focus on query readability and not identifying flaws.
* According to [13], ERD-like visualization doesn’t have a big impact on the mental workload for understanding SQL queries.

2.6.3 Visualize the Actual Execution Statistics

These techniques give an intuitive understanding of the query and provide users with a much more granular understanding of the DBMS.   
  
Perfopticon [12] helps to understand the queries' bottlenecks in distributed databases by visualizing the execution plan and how different operation behaved across multiple servers. Perfopticon support several types of visualization, some focus on how the physical plan and operations on different fragments and another that describe query’s characteristics for a specific fragment.

Although visualizing the actual execution statistics can be useful, they have the following cons:

* Existing solutions focus on identifying bottlenecks for a single query only.
* Might become resource-heavy for big data use-cases.

2.6.4 Visualize the Optimizer Statistics

These techniques focus on visualizing the behavior of modern optimizers. Database optimizers produce several execution plans for each query, and the ability to compare between optimal execution plans and visualize it intuitively.   
  
Picaso [13] takes as input a query, and an optimizer, and generates several visualizations that help understand characterize the behavior of the optimizer. We can understand and compare how long does the optimizers compilation time took or how good the plans they generated.   
  
In the next section, I am going to cover a comparison of the approaches for debugging SQL and identifying flaws in SQL queries.

##### 2.7 Approaches Comparison

In the previous sections (2.3, 2.4, 2.5 and 2.6), we covered several approaches that can help identify and fix flaws when using DBMS. In this section, we are going to cover the conclusions of a survey on tools for debugging database queries [14].

The study conducted included 20 participants, including 6 undergraduate students, 4 graduate students, and 10 industry professionals, and covered all the approaches I listed above. Awareness of database debugging techniques was low, as only 4 out of 20 knew they even exist.

The participants stressed that DBMS errors and execution plans are hard to interpret and provide little help understanding the errors. As a result, the most common technique the participants used for debugging flawed queries was trial and error. They wrote their query, and then manually review the raw results from the DBMS, a wasteful and error-prone approach. To find flaws in complex and nested queries, they tend to simplify the query and divide it into several components, which make things even more wasteful.

The participants found static analysis and the debugger were less intuitive and effective. They found the following two techniques to be useful to mitigate these problems listed above:

* **Using visual aids** – many of the participants believe visual aids help to identify flaws in their queries. Most suggestions involve displaying intermediate results of the queries, which highlight the trace of where certain tuple came from. It is important to know that some visualization techniques, such as generating ER diagrams were not found helpful.
* **Unexpected result indicator** – all the participants mentioned that summary statistics can benefit them when the query result was unexpected. Even though statistics seem to be simple, none of the tools they tried to support these. Some of the metrics they include:
  + The number of rows returned.
  + Presence of NULL entries.
  + Presence of duplicates.
  + The number of rows returned for a query sub-expression.

We have covered and compared the techniques to identify flaws in SQL queries, in particular, we covered some techniques that try to identify and optimize the query bottlenecks. The techniques that I have covered focus on optimizing a single query, but in the real-world queries affect one another.   
  
In the next section, I will review relevant related work on multi queries optimizations.

##### 2.8 Multi Query Optimization

As organizations become more data-driven, having complex queries is the new standard. To speed up a complex query, some optimizations to the database or the query itself may be needed. But optimization that may benefit one query may have a negative effect on others. For this reason, Multi-query optimization (MQO) is hard.

To the extent of my knowledge, all the related work focuses on the optimizer itself, and how to make it effective for the MQO problem. In those techniques, the optimizer can evaluate several queries together, such that the queries can share state and save repeated computation. Volcano [15], introduce several cost-based heuristic algorithms that decide what sub-expressions should be materialized and shared using DAGs. Rule-based framework [16], that incorporates a set of new abstractions, that allow integrating new and existing MQO techniques through the use of transformation rules.  
  
Unfortunately, the MQO problem is much broader than how to share state effectively across multiple queries. When a user optimizes his query, it may have a negative effect on others, such behavior will not be handled in the optimizer. For example, an index will make a scan faster but can have a negative effect on updating a relation.   
  
For this reason, the ability to visualize multiple queries together is valuable. The following work [17] visualizes the result of multiple queries together, to allow easy comparison. But this technique focuses on data exploration and not on identifying bottlenecks and flaws like QueryFlow.

In section 2.6 we covered visualization techniques to identify problems in SQL queries. Unfortunately, these techniques cannot compare statistics of sub-expression characteristics, while keeping the sub-expressions hierarchy clear.

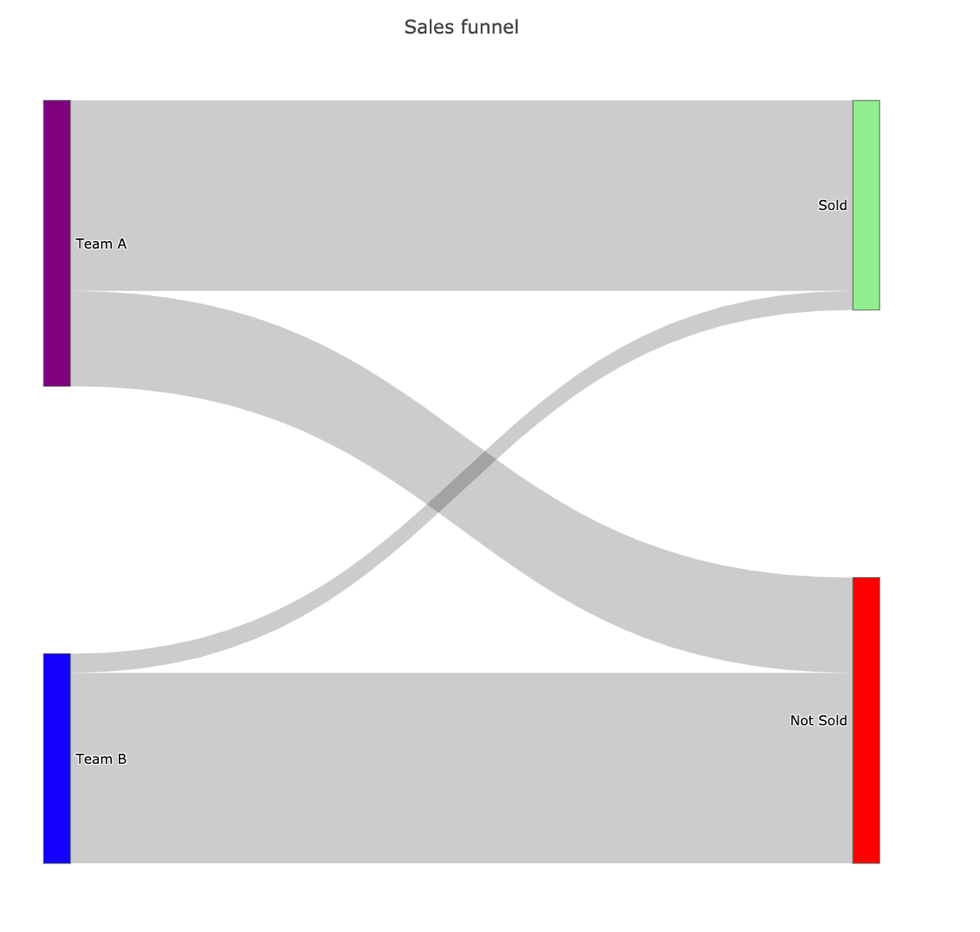
Sankey-diagram is a visualization technique that can mitigate it and I am going to cover it in section 2.9

##### 2.9 Sankey Diagrams

Sankey-diagram [1] is a visualization technique to display any kind of measurable flow. Sankey-diagram is a graph representation with specific characteristics on the nodes and edges. The nodes represent the entity and are visualized as a colored rectangle. And the links represent a measurable metric and are visualized as an edge with a width proportional to the metric measure.

I will show through a naïve example, the merits of the Sankey-diagrams. In our example, we have two sales teams, and we want to see which sales team performs better. We will create a Sankey-diagram that represent our problem:

* We will have four nodes: one for each team and another two to represent if a deal was closed (sold) or not.
* We will have four edges: from each node representing a team, we will have an edge to both deals that are sold and deals that are not sold, and the thickness of the edges will represent the number of deals.

The Sankey-diagram that represents our example can be seen in Figure 8.

**Figure 8-** Sankey Diagram of Sales Funnel of both teams

We can see in our example in Figure 8 that Team A did a much better job than team B.

* Team A sold a lot more than Team B as the width of the edge between the “Team A” node and the “Sold” edge is much thicker than the edge between the “Team B” node and the “Sold” node.
* Team A conversion is better than Team B as most of the width of the edges from the “Team A” node is connected to the “Sold” as opposed to the “Team B” node.

The Sankey diagram allows us to show extra information if needed. We can show the actual value which represents our link width (our measurable metric) and additional information if needed, by hovering the edge. This can be seen in Figure 9.

****

**Figure 9-** Additional information pop up when hovering an edge

From the naïve example I have covered, we can see the following merits of Sankey-diagrams

* Easy to display any measurable flow of data.
* We can distinguish between different types of nodes using colors.
* We can distinguish between different types of edges using colors.
* We can add additional metadata for each edge when we hover over it.

There are many studies in the academy that uses Sankey to represent resource utilization. For example, understanding the utilization of energy ecosystems [18] across different devices and sectors using Sankey diagram. In Figure 10 we can see that Liquid fuel is a big portion of all the fuels and its highly used on the road by diesel engines and so on.

Diagram

Description automatically generated   
 **Figure 10-** Sankey diagram of the energy ecosystem (Taken from paper 18)

In addition, since Sankey popularity is increasing, optimizing the Sankey-diagrams has become a research topic in its own right. In particular, techniques that enable a nicer layout, whether in terms of the number of crossing edges or their thickness. For example [19] uses integer linear programming to approximate this. It models the problem by defining variables, constraints, and an objective function to approximate NP-hard problems within a reasonable time.

##### 2.10 Summary

In this chapter, we provided an overview of the most common flaws DBMS users face, and several approaches to tackle those.

The survey in section 2.8 shows that out of those approaches visualizing the execution plans seems to be the most promising and intuitive for users. But the existing solutions lack the following properties:

* Display intermediate results of the query.
* Summary statistics like the number of rows returned for a query sub-expression.
* They focus on performance optimization or query readability.
* They visualize either the logical execution plan or the actual execution statistics, but never both.
* They visualize one query only.

We created QueryFlow to mitigate those lacking properties, by providing a flexible tool with the right visualization in mind. Since SQL queries have interesting measurable statistics for their intermediate sub-expressions, Sankey-diagram is the natural pick to understand the behavior on a sub-expressions granularity due to the merits we covered in section 2.9.  
  
QueryFlow is like a swiss-knife, it allows to visualize both logical execution plan and actual execution statistics for one or more queries. QueryFlow can help to identify both cardinality and performance problems in the same manner, by visualizing the relevant statistics.

In the next chapter, we will see QueryFlow design and understand how it works internally. In the following chapter, we will see the use cases QueryFlow can support.

# Chapter 3: QueryFlow Design

Identifying flaws in complex queries and bringing them to perfection is challenging. One of the most prominent techniques to achieve this is visualizing queries’ execution plans.

To help users identifying their problems and solve them we created QueryFlow. QueryFlow visualizes the query execution using the Sankey diagram, a technique that allows one to illustrate complex processes, with a focus on a single aspect or resource that you want to highlight.

QueryFlow will bring the following improvements to current tools:

* Capabilities to visualize both the logical and execution structure.
* Represent query characteristics as part of the visualization.
* The ability to visualize multiple queries.

The rest of this chapter is structured as follows:

* Section 3.1 provides a bird's eye overview of QueryFlow design.
* Section 3.2 provides an overview of QueryFlow’s parsing component.
* Section 3.3 provides an overview of QueryFlow’s enrichment component.
* Section 3.4 provides an overview of QueryFlow’s visualization component.
* Section 3.5 provides a detailed example of QueryFlow’s components.

3.1 QueryFlow design  
In this section, we give an overview of QueryFlow’s design and how it is built. We design QueryFlow support visualizing both the logical execution plan and the actual execution statistics.   
  
This allows QueryFlow to identify the following families of flaws:

* Cardinality issues- finding errors due to WHERE*, JOIN, UNION, DISTINCT, HAVING* clauses by utilizing the *Actual Rows* or *Planned Rows.*
  + Query result with zero entries (“The empty answer problem”).
  + Filters that remove no entries (“Why and Why not problem”).
  + Joins relation that returns an unexpected number of entries (“Why and Why not problem”).
  + Identifying duplication.
* Queries bottlenecks – finding the queries’ bottlenecks, for a single query or multiple queries.
* Optimizer problems – by comparing the optimizer expectations and reality for sub-expressions.

We designed QueryFlow with minimal requirements from the database system. This makes it easier to generalize beyond one database, as it only requires either the query logical execution plan or the query actual execution statistics.   
  
QueryFlow process consists of the following steps (Figure 11):

1. **QueryFlow parsing**- First we get the execution plans for our input queries using the *EXPLAIN* clause. After we have the relevant execution plans, we can start the parsing phase. We will traverse each one sub-expression to a concise structured representation, as similar sub-expressions can be grouped, and irrelevant sub-expressions are filtered. More details can be found in section 3.2.
2. **QueryFlow enrichment –** The parsed representation of a query is useful in its own right, but it lacks some important statistics, and some statistics are not in the right granularity. For these reasons, we are going to infer extra statistics using the existing ones in the enrichment phase. This will allow us to represent the query characteristics in a much more intuitive way later in the visualization phase. More details can be found in section 3.3.
3. **QueryFlow visualization**- We start the visualization process by transforming the enriched representation of our query into a representation that is more suitable for Sankey-diagrams. Then we check what colouring heuristics we should apply to each sub-expression. Lastly, we visualize the enriched structured representation using Sankey-diagrams to emphasize the query’s characteristics. More details can be found in section 3.4.

Diagram

Description automatically generated

**Figure 11-** QueryFlow Design

In this work, I have used PostgreSQL, but for QueryFlow is written with extendibility in mind. To onboard a new database, we need to implement a parsing function for each relational operation, which is a small fraction of the parsing phase only.

##### 3.2 QueryFlow Parsing

The parsing stage begins with getting the execution plans for our queries. To get the execution plan we use either the *EXPLAIN* clause or the *EXPLAIN ANALYSE* clause*.* The difference between *EXPLAIN* and *EXPLAIN ANALYSE* is that the first only give us estimated statistics about a query and the second executes the query and provide both estimated statistics and real statistics.   
  
After we execute our queries with these clauses, we get the execution plans with relevant and useful statistics for each sub-expression of the query. There are various statistics in the execution plan that can help understand the behavior of the query. For example, there is a statistic that corresponds to the number (or an estimation) of records a sub-expression holds, another useful statistic is the execution time of the sub-expression (or an estimation).   
  
To later incorporate these statistics into the Sankey-diagram, we need to prepare the statistics for each sub-expression as a graph. To do so, we need to add to each sub-expression which expression is its ancestor, and which is its child, this can be achieved by recursively navigating the sub-expressions and build a tree-like structure.   
  
When we are working with multiple queries, there are additional few steps. First, on some databases, we need to clean the cache between each query execution to represent the query execution in the best way possible. Secondly, we want the ability to represent similar sub-expression by the same node (and have multiple edges). To do so, we specify a hash function that indicates whether two sub-expressions from different queries are the same logically and provide a sign for that.

When we parse the execution plan, we can use heuristics to have a more compact representation of our query. For example, some operations like *HASH* don’t have any effect on the number of rows, and we can “skip them” when we parse the execution plan identifying cardinality flaws.   
  
The parsing algorithm can be seen in section 3.4.3 for a more fluent read (Figure 19).

The parsed execution plan is useful on its own, but it lacks some important statistics, and some statistics are not in the right granularity. In the next section, we will how QueryFlow’s enrichment phase can mitigate it.

##### 3.3 QueryFlow Enrichment

After the parsing phase, we have parsed execution plans, one for each query. Unfortunately, it lacks some important statistics, and some are not in the right granularity. To mitigate it, we are going to infer extra statistics in the enrichment phase.

The enrichment phase is valuable for enriching both the execution plan from *EXPLAIN* clause or *EXPLAIN ANALYSE* clause. We are going to infer new statistics from the existing ones, some of the more prominent statistics we are adding are:

* **Missing statistics in sub-expression granularity-** many statistics are cumulative and include aggregations of the ancestor sub-expressions. Since we want to identify the sub-expression that caused flaws in our query, we want the statistics at a more granular level (the sub-expression level). For example, when we are looking for performance bottlenecks in our query, we want to be able to easily identify the sub-expression with the highest duration. Unfortunately, we only get the *total\_time* until the sub-expression was executed (included). So, we need to calculate the sub-expression duration by subtracting the *total\_time* of the current node from the largest *total\_time* of its ancestors, which will give us the sub-expression duration as we wanted.
* **Percentage statistics** – having certain statistics as a percentage can be very useful. This becomes critical when we compare two metrics measured by different units. For example, if we want to compare the optimizer estimation and the actual execution time, we must compare them as a percentage as they work with different units that are not comparable.
* **Redundant operations**- many queries have redundant parts; the result is the same with or without them. Finding these redundant operations can help us to improve the query performance, as it will do fewer operations. For example, a non-effective operation will be a *distinct* operation that filters nothing, which can help us improve the query performance as a *distinct* operation tend to be very heavy.
* **Human-readable representation**- When we visualize each of our sub-expressions, we want to be able to represent it in a readable and understandable manner. We will transform each sub-expression label to be its’ relational representation. For example, instead of representing a join between two tables as follows *T1 JOIN T2,* we will present it as *T1 ⋈ T2.*

Since the actual execution statistics and planned execution plan only differ in the metrics themselves, apart from the redundant operation, all the enrichment types we described above are relevant for both types. The reason the redundant operation is useful for logical execution plan is that it’s an estimation that will bring more false positives and will lure the user from the real problems.

The enrichment algorithm can be seen in section 3.5.4 for a more fluent read (Figure 21).

In the next section, we will how QueryFlow’s visualization phase takes the enriched execution plan and visualize it in an intuitive way.

3.4 QueryFlow Visualization

A quick reminder, Sankey-diagrams allow one to illustrate complex processes, with a focus on a single aspect or resource that you want to highlight. Since SQL queries have interesting statistics for their intermediate sub-expressions such as the cardinality and the duration we can understand how the data “flow” in the query.

After we parsed and enriched the execution plan we want to transform it into a more suitable representation for Sankey-diagrams. We want the nodes to represent relational sub-expression. While edges, will represent a parent-child relationship, aka the operator that was applied to the relations and width proportional to a measurable statistic, additional details of the operator are provided when hovering an edge.

To do so, we transform the tree-structured execution plan to a tabular representation, where we have the *source* (represents the current sub-expression), *target* (represents one direct ancestor), *value* (represents the metric value we compare), *variable* (represents the metric name we compare), and *label* (the text that represents the current sub-expression). A detailed example can be seen in section 3.5.5 (Figure 24).  
  
In cases where we want to visualize multiple metrics, we will represent them as two edges between the same nodes with different brightness. This will require an extra step, of pivoting the metrics to be represented in different rows. The rows will have the same *source*, *target*, *label* and *value* will differ in the *variable* which will be different, meaning they will be two different edges between the same nodes.

QueryFlow support advance and configurable coloring mechanism for both edges and nodes.  
QueryFlow’s nodes represent relational sub-expression we can be colored by sub-expression type, for example, we can indicate that all the *Join* operations will be blue. By default, all the nodes are colored in black as it can be quite overwhelming due to the number of different operations.

QueryFlow’s edges represent measurable statistics like cardinality or duration. To emphasize parts of the query that might need some special attention we will add coloring heuristics to the edges and nodes of the Sankey-diagram. A detailed example can be seen in section 4.1.  
  
The first set of heuristics is the edges and indicate a potential flaw in the sub-expressions :

* When a relation cardinality is zero, this can help us find the problematic sub-expression we need to rewrite to fix cardinality issues.
* When an operation is redundant, this can help us improve the queries’ performance by removing operations that don’t change the actual output.

The second set of heuristics is on the edges as well and helps differentiate between different entities:

* We want to be able to distinguish between multiple queries. We represent each of them in a different color by randomly assigning each query a color. A detailed example can be seen in section 4.6 .
* We want to be able to distinguish between multiple metrics of the same query. We represent each of them in the same colors with different saturation. A detailed example can be seen in section 4.5.

The last set of heuristics are on the nodes and they help to distinguish between different types of relational operations, but it disabled by default.

The visualization algorithm can be seen in section 3.5.5 for a more fluent read (Figure 23).

In the next section, we cover a detailed example of QueryFlow execution, to understand what QueryFlow does behind the scenes in each of its steps.

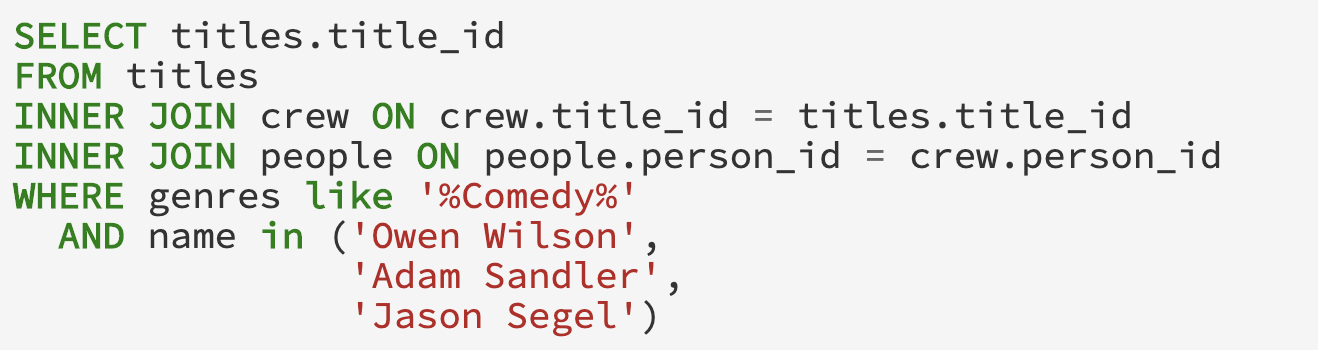
##### 3.5 QueryFlow Detailed Example

3.5.1 Example Definition  
  
Through chapter 3 and chapter 4, I am going to use the IMDB dataset. The dataset contains a total of eleven tables, but we only use *titles, crew, people* and, *genre* tables. The tables and their relations can be described in Figure 12.  
Diagram

Description automatically generated

**Figure 12-** IMDB Schema

We introduce a simple example, that is rich enough to illustrate the gist of the QueryFlow and how it works under the hook. From the input query to the parsing phase through the enrichment phase and finally the visualization phase.  
  
In our example, we want to answer the following question, “What movies are recommended for me? given that I love comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”. This question is equivalent to the following SQL query defined in Figure 13.



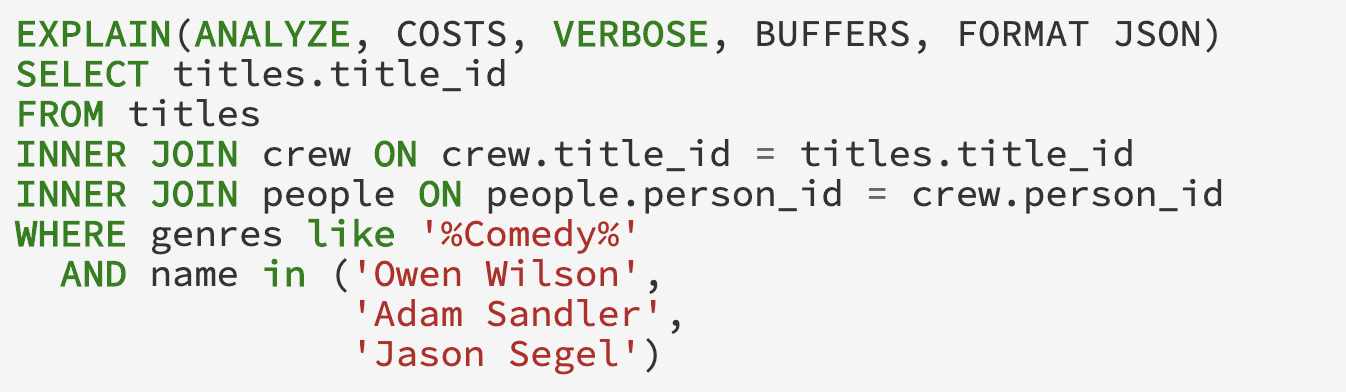
**Figure 13-** Detailed Example’s Query

3.5.2 Getting the Database Execution Plan  
  
After we are given the query in Figure 13, we will modify the query by adding the *EXPLAIN* prefix to the beginning of the query.   
  
This will give us a new query that will return either the logical execution plan or the actual execution statistics (the query is executed) depending on the input. We will show the modified queries to get both the logical execution plan and the actual execution statistics.   
  
In both cases, we will add to *EXPLAIN* clause with the following attributes “*COSTS, VERBOSE, BUFFERS, FORMAT JSON”* to get additional statistics in a *JSON* format. If we want to get the logical execution plan, we will add the following to our original query “EXPLAIN(*COSTS, VERBOSE, BUFFERS, FORMAT JSON)”* and we will get the query in Figure 14.

Text

Description automatically generated

**Figure 14-** Detailed Example’s Explain Query  
  
If we add the *ANALYZE* keyword, we will get the actual execution statistics and we will get the query in Figure 15.



**Figure 15-** Detailed Example’s Explain Analyze Query

When executing the modified query in Figure 14 and Figure 15 we will get the execution plans. Both execution plans are a huge nested *JSON*  that includes statistics, on a sub-expression granularity. Since it’s hard to cover such a huge *JSON,* for readability purposes I will omit most of the statistics.   
  
The *JSON* representation of the actual execution statistics can be seen in Figure 16.

Timeline

Description automatically generated  
 **Figure 16-** Detailed Example’s Explain Analyze Results   
Just for the sake of comparison, the logical execution plan *JSON* can be seen in Figure 17 and it will have the same structure as the actual execution statistics (Figure 16) apart from some missing statistics like the *Actual Rows* and *Total Time*.   
Text

Description automatically generated  
 **Figure 17-** Detailed Example’s Explain Results   
  
We can see in both Figure 16 and Figure 17 that the logical and actual execution statistics are a nested *JSON* which represents the relational sub-expressions. These sub-expressions describe the nature of part of the query and how it was (or going to be) executed.

Each sub-expression has different keys depending on the type of the sub-expression and whether it’s the logical execution plan or the actual execution statistics. There are a lot of different statistics and information, but generally, it can be divided into the following groups:

* **Node Type –** the type of relational operation it is whether it’s a scan, a join, or other relational operation.
* **Plans –** a list of direct ancestors for the current sub-expressions.
* **Measurable metric –** a number that represents a certain characteristic of the operation. For example, *Actual Rows* represent the number of rows returned by the operation.
* **Additional Information –** a text which helps us understand which part of the query it is. For example, when we use *Seq Scan* we need to know which relation, and for that, we got *Relation Name.*

A more intuitive way to think about both the logical execution plan and the actual execution statistics is as a tree structure. QueryFlow works in the same manner for both the logical execution plan and the actual execution statistics. From this step forward, I will demonstrate the actual execution statistics only.

You can see the equivalent to our actual execution statistics from Figure 16 in a tree representation (with parts of the information) in Figure 18.  
**Diagram

Description automatically generated**

**Figure 18-** Detailed Example’s Tree Representation of Execution Plan

3.5.3 QueryFlow’s Parsing Phase

Now we going to parse the *JSON*, by recursively visiting the sub-expression ancestors. As we said, the ancestors are specified by the *PLANS* key, and a sub-expression is terminal (has no ancestor) if the *PLANS* key is empty. The parsing algorithm can be found in figure 19:  
Text, letter

Description automatically generated   
 **Figure 19-** Parsing Algorithm

We start parsing our example in Figure 18 and emphasize the parsing capabilities of removing irrelevant sub-expression, we will focus in this example on finding cardinality issues by using the *Actual Rows* metrics.

By doing so we will get the following ancestors hierarchy:

1. The *Gather* operation (collect relevant records from the workers) has one ancestor, the *Hash-Join* between *titles*, *crew* and *people*.
2. The *Hash-Join* operation has two ancestors, the *Hash* operation and the *Hash Join* between *crew* and *people* operation.
3. The *Hash* operation has one ancestor, the *Seq Scan* of *titles*. Since we are looking for cardinality issues, and the *HASH* operator does not affect it, we can skip the hash and continue with the *Seq Scan* parsing. Since the *Seq Scan* has the filter key inside it, when we parse this operation, we are going two splits it into two logical operations. This later will allow us to better understand the query.
   1. *Seq Scan\** on the titles which represents the titles after the filter, and it will instead of the *Seq Scan* in the ancestor hierarchy.
   2. *Seq Scan* on the titles which represents the titles before the filter and will be the ancestor of the new *Seq Scan\* operator*.
4. The *Hash-Join* operation has two ancestors, the *Seq Scan* of crew and the *Hash* operation.
5. The *Hash* operation has one ancestor, the *Seq Scan* on people. Since we are looking for cardinality issues, and the *HASH* operator does not affect it, we can skip the hash and continue with the *Seq Scan* parsing. Since the *Seq Scan* has the filter key inside it, when we parse this operation, we are going two splits it into two logical operations. This later will allow us to better understand the query.
   1. *Seq Scan\** on the people which represents the people after the filter, and it will instead of the *Seq Scan* in the ancestor hierarchy.
   2. *Seq Scan* on the people which represents the people before the filter and will be the ancestor of the new *Seq Scan\* operator*.
6. the *Seq Scan* of *crew* has no ancestor*.*

Again, I will show the actual execution statistics after parsing as the tree, and it can be seen in Figure 20.

Diagram

Description automatically generated

**Figure 20-** Detailed Example’s Tree Representation of Execution Plan After QueryFlow’s  
 Parsing phase.  
  
The differences between Figure 18 and Figure 20 are:

* We have separated *Scan* sub-expressions into two, one for the *Scan* and one for the filter for both the *People* relation and the *Title* relation.
* We can drop sub-expressions that don’t change the cardinality like the *Hash* sub-expression of the *People* relation and the *Title* relation*.*

3.5.4 QueryFlow’s Enrichment Phase

Now we got to the enrichment phase, as we still lack some can relevant information. We will enrich our execution plan in Figure 20 using the algorithm in Figure 21.

Text

Description automatically generated

**Figure 21-** Enrichment Algorithm

The algorithm iterates the ancestor’s hierarchy as a BFS, and the iterations go as follows:

1. We run on the node which represents the *People* sub-expression.
   1. **Missing statistics in sub-expression granularity-** since this operation has no ancestors, no additional work is required here*.*
   2. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *People actual\_duration.*
   3. **Redundant operations**- it’s not a redundant operation because it’s a *Scan*.
   4. **Human-readable representation**- *People.*
2. We run on the node which represents the *People\** sub-expression.
3. **Missing statistics in sub-expression granularity-** We calculate the *actual\_duration* by subtracting the *People total\_time* from *People\* total\_time.*
4. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *People\* actual\_duration.*
5. **Redundant operations**- it’s not a redundant operation because rows were filtered.
6. **Human-readable representation**- *People\*.*
7. We run on the node which represents the *Crew* sub-expression.
   1. **Missing statistics in sub-expression granularity-** since this operation has no ancestors, no additional work is required here*.*
   2. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *Crew actual\_duration.*
   3. **Redundant operations**- it’s not a redundant operation because it’s a *Scan*.
   4. **Human-readable representation**- *Crew.*
8. We run on the node which represents the *People\* ⋈ Crew* sub-expression.
9. **Missing statistics in sub-expression granularity-** We calculate the *actual\_duration* by subtracting the maximum between *People\* total\_time* and *Crew total\_time* from the *People\* ⋈ Crew total\_time.*
10. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom People*\* ⋈ Crew actual\_duration.*
11. **Redundant operations**- it’s not a redundant operation because it’s a *Join*.
12. **Human-readable representation**- since it’s an expression with multiple ancestors we calculate the label to represent this sub-expression, we replace the JOIN label with People*\* ⋈ Crew.*
13. We run on the node which represents the *Title* sub-expression.
    1. **Missing statistics in sub-expression granularity-** since this operation has no ancestors, no additional work is required here*.*
    2. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *Title actual\_duration.*
    3. **Redundant operations**- it’s not a redundant operation because it’s a *Scan*.
    4. **Human-readable representation**- *Title.*
14. We run on the node which represents the *Title\** sub-expression.
15. **Missing statistics in sub-expression granularity-** We calculate the *actual\_duration* by subtracting the *Title total\_time* from *Title\* total\_time.*
16. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *Title\* actual\_duration.*
17. **Redundant operations**- it’s not a redundant operation because rows were filtered.
18. **Human-readable representation**- *Title\*.*
19. We run on the node which represents the *People\* ⋈ Crew ⋈ Title\* sub-expression*.
20. **Missing statistics in sub-expression granularity-** We calculate the *actual\_duration* by subtracting the maximum between *Title\* total\_time* and *People\* ⋈ Crew total\_time* from the *People\* ⋈ Crew ⋈ Title\* total\_time.*
21. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *People\* ⋈ Crew ⋈ Title\* actual\_duration.*
22. **Redundant operations**- it’s not a redundant operation because it’s a *Join*.
23. **Human-readable representation**- since it’s an expression with multiple ancestors we calculate the label to represent this sub-expression, we replace the JOIN label with *People\* ⋈ Crew ⋈ Title\*.*
24. We run on the node which represents the *Gather sub-expression*.
25. **Missing statistics in sub-expression granularity-** We calculate the *actual\_duration* by subtracting *People\* ⋈ Crew ⋈ Title\* total\_time* from the *Gather total\_time.*
26. **Percentage statistics** – We calculate the *actual\_duration\_pct* by dividing the query execution timefrom *Gather actual\_duration.*
27. **Redundant operations**- it’s not a redundant operation because it’s a *Gather*.
28. **Human-readable representation**- *Gather* does not require new representation*.*

The result of the enriched representation of our example can be seen in a tree structure in Figure 22.

Diagram

Description automatically generated

**Figure 22-** Detailed Example’s Tree Representation of Execution Plan After QueryFlow’s  
 Enrichment Phase.  
  
The differences between Figure 20 and Figure 2 are:

* We have nice labels for each our nodes.
* We have statistics in the right granularity like *Duration* as oppose to *Total Time.*
* We have statistics as percentage as well.

3.5.5 QueryFlow’s Visualization Phase

Now that we got our execution plan parsed and enriched, we are ready to start the visualization phase. We will transform our tree representation to a more friendly representation for Sankey-diagrams.

The table representation includes the following information (and much more):

* **source/ target –** describe the ancestors’ hierarchy of a relational operator. The *source* column is an identifier of the current node and the *target* is an identifier of one ancestor of the current node.
* **operation\_type –** isthe unparsed node type (*Node Type)* in the execution plan.
* **label –** logical representation of the operation type, this will allow us to group similar operators like *Hash Join* and *Merge Join* on the same relation.
* **label\_metadata –** additional information of an operator, that can be useful for the observer. Each node type (*Node Type)* has different useful information. For example, an important piece of information for a *Hash Join* is the join condition (*Hash Cond).*
* **actual\_rows –** is one of the metrics we want to measure that represents the sub-expression cardinality.
* **actual\_duration –** is one of the metrics we want to measure that represents the sub-expression execution time.

The table representation for our enriched execution plan can be seen in Figure 23.

Table

Description automatically generated

**Figure 23-** Detailed Example’s Table Representation of Execution Plan After QueryFlow’s  
 Enrichment Phase.

Figure 23 represents each sub-expression as a row. In our example the sub-expressions rows can be described as follows:

1. The node representing *People* is a table scan with 3,446,261 rows and has an edge to *People\**.
2. The node representing *People\** has 3 rows and has an edge to *People\* ⋈ Crew*.
3. The node representing *Crew* has 13,651,901 rows and has an edge to *People\* ⋈ Crew*.
4. The node representing *People\* ⋈ Crew has* 565 rows and has an edge to *People\* ⋈ Crew ⋈ Title*.
5. The node representing *Titles* has 489,076 rows and has an edge to Titles*\**.
6. The node representing *Titles\** has 489,076 rows and has an edge to *People\* ⋈ Crew ⋈ Title*.
7. The node representing *People\* ⋈ Crew ⋈ Title\* has* 558 rows and it is a terminal node.

To create the table representation we will use the visualization algorithm in Figure 23.

Text

Description automatically generated

**Figure 23-** Visualization Algorithm

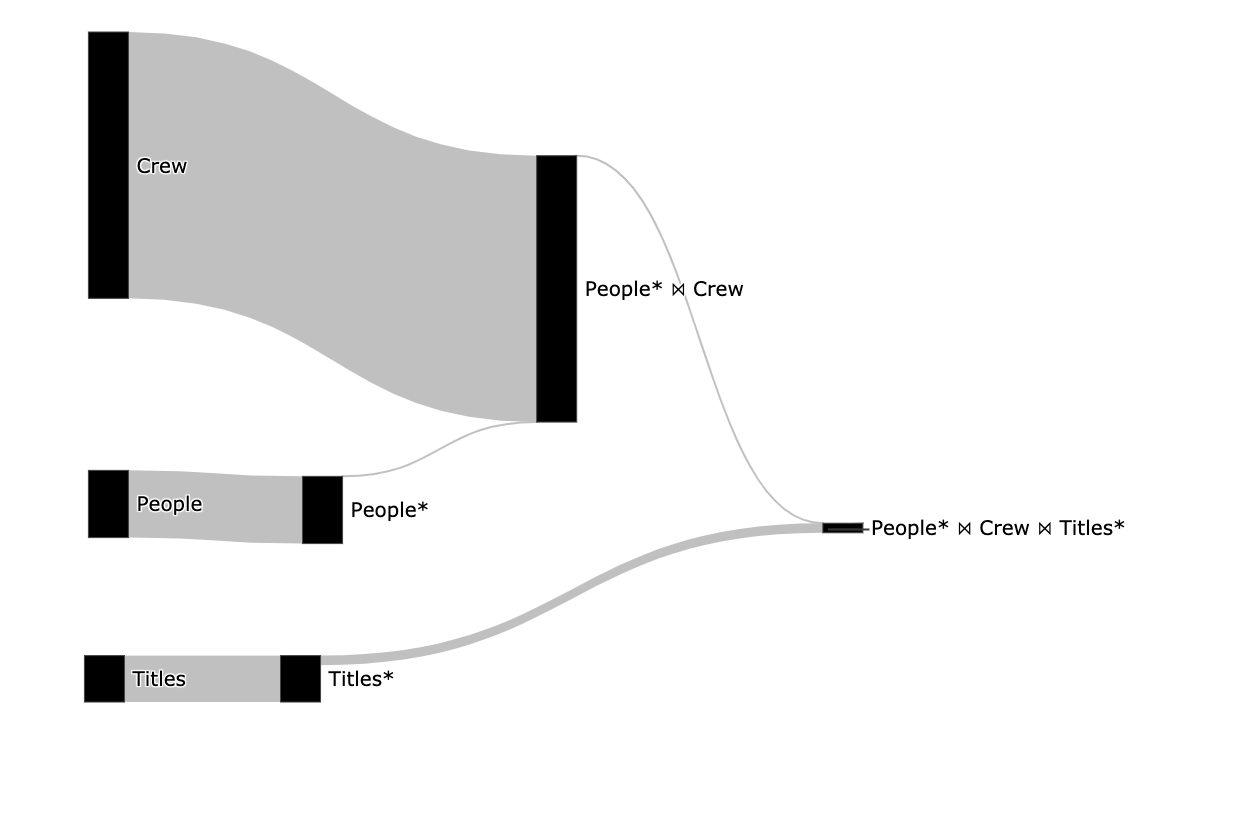
Add suffix

Before we are going to plot our table representation, we will check if special coloring heuristics are needed. We will check if any coloring heuristics are met:

* Sub-expression has zero cardinally (red edge).
* Sub-expression is redundant (orange edge).
* We are investigating multiple queries.
* We are investigating multiple statistics.
* We want different coloring for relational operations.

Since none of the coloring heuristics are met will stick to the default coloring scheme and we will go with the defaults (gray edges and black nodes).

We are going to visualize our table representation as a sankey-diagram. The nodes will represent the sub-expression label, while edges will represent a parent-child relationship, and they will be proportional to the cardinality (*actual\_rows)*. Additional details of each sub-expression are can be seen when hovering over an edge. Figure 25 describe the cardinality Sankey diagram for our query.



**Figure 25-** QueryFlow Cardinality Sankey diagram for our query

After we created the visualization, we can see the sub-expression hierarchy and the cardinality of each sub-expression, by how thick an edge is. Now we can understand why splitting the *Seq Scan* operator is so valuable, I can understand how many rows the original relations had (*titles* and *people)* and the cardinality we got after the filter (*titles\** and *people\*)*.  
  
From figure 25 we can understand a lot about our query, including the following:

* The filter on the people relation is not redundant as the edge from it is thinner, and an index might be able to improve performance.
* The filter on the title relation is not redundant as the edge from it is thinner, and the index might be able to improve performance.
* No sub-expression is empty (has no rows) as there is no edge colored in red.
* We can understand the size of the relations, the crew is by far the biggest relation.

We can visualize a Sankey diagram for duration as well as can be seen in Figure 26.

Diagram

Description automatically generated with medium confidence

­­­­ **Figure 26-** QueryFlow Duration Sankey diagram for our query

In the next chapter, we will see the use cases QueryFlow can and can’t support.

# 

# Chapter 4: QueryFlow Use-cases

In the last chapter, we saw how QueryFlow is designed and went through a detailed example. In this chapter, we will see the use cases and flaws QueryFlow can identify.

QueryFlow is an enabler over the logical execution plan or the actual execution statistics. For this reason, it allows identifying a lot of different flaws, by visualizing the relevant statistics.

The rest of this chapter is structured as follows:

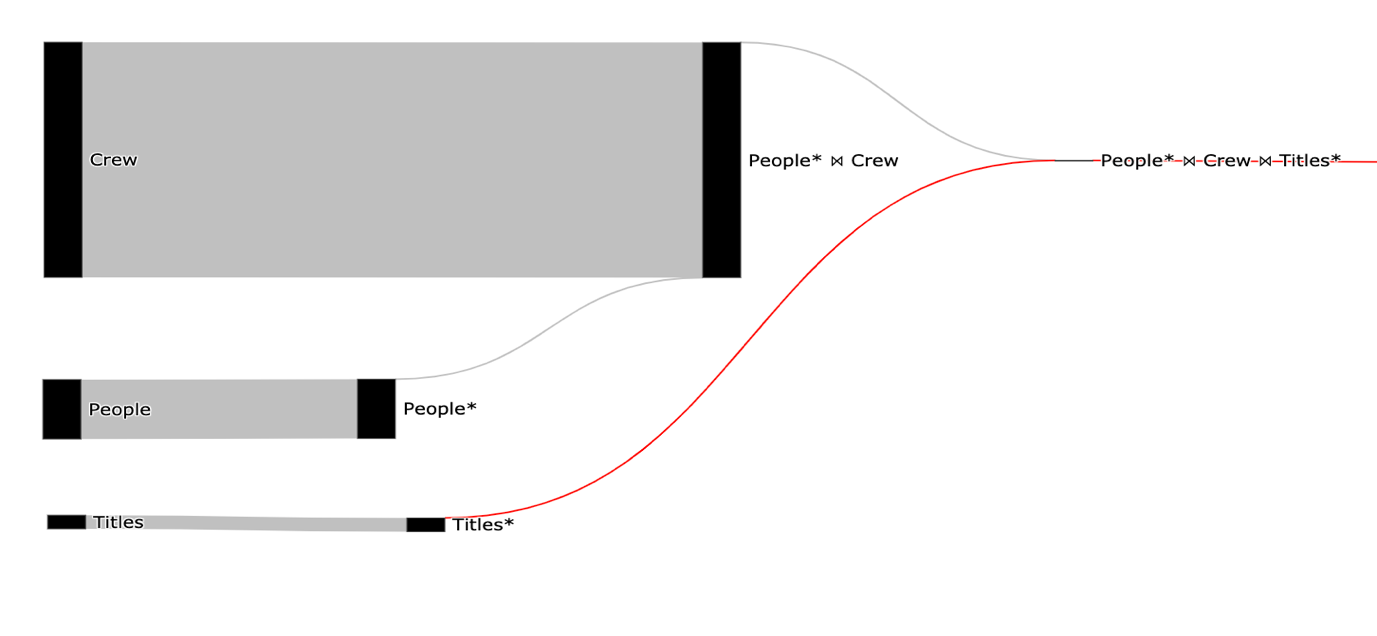
* Section 4.1 provides an example of identifying missing records using QueryFlow.
* Section 4.2 provides an example of identifying ineffective operations using QueryFlow.
* Section 4.3 provides an example of identifying duplicated entries using QueryFlow.
* Section 4.4 Identifying performance bottlenecks in a single query using QueryFlow.
* Section 4.5 provides an example of identifying flaws in the optimizer itself using QueryFlow.
* Section 4.6 provides an example of Identifying performance bottlenecks across multiple queries.
* Section 4.7 explains when QueryFlow won’t help to identify flows in your SQL queries.

4.1 Identifying missing recordsProblems related to missing records are common and finding them tends to be hard for non-experts. Using QueryFlow we can visualize the cardinality (*actual\_rows)* of the query’ sub-expressions and find the first sub-expression that resulted in an empty result. This problem can be caused due to either *WHERE, JOIN, UNION,* or *HAVING* clauses. In a similar way we can solve the why and why not problem.

We will use the same example as in chapter 3. However, we will introduce a small bug in our SQL query that will cause the query result to be empty. The modified query can be seen in Figure 27.

****

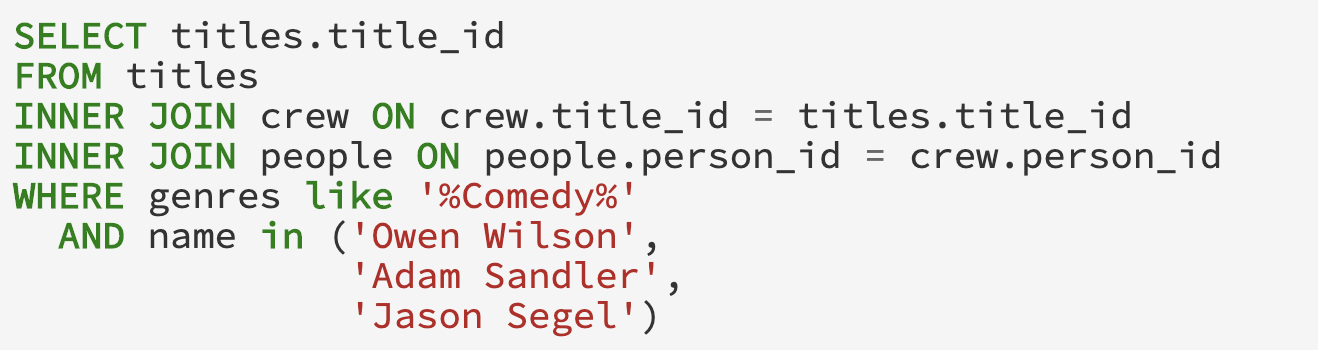
**Figure 27-** Identifying missing records query  
We are going to use QueryFlow to identify the empty results (will have a red edge) and from which sub-expression it was originated. The corresponding Sankey in Figure 28 represents the cardinality of our query’s sub-expressions.

****

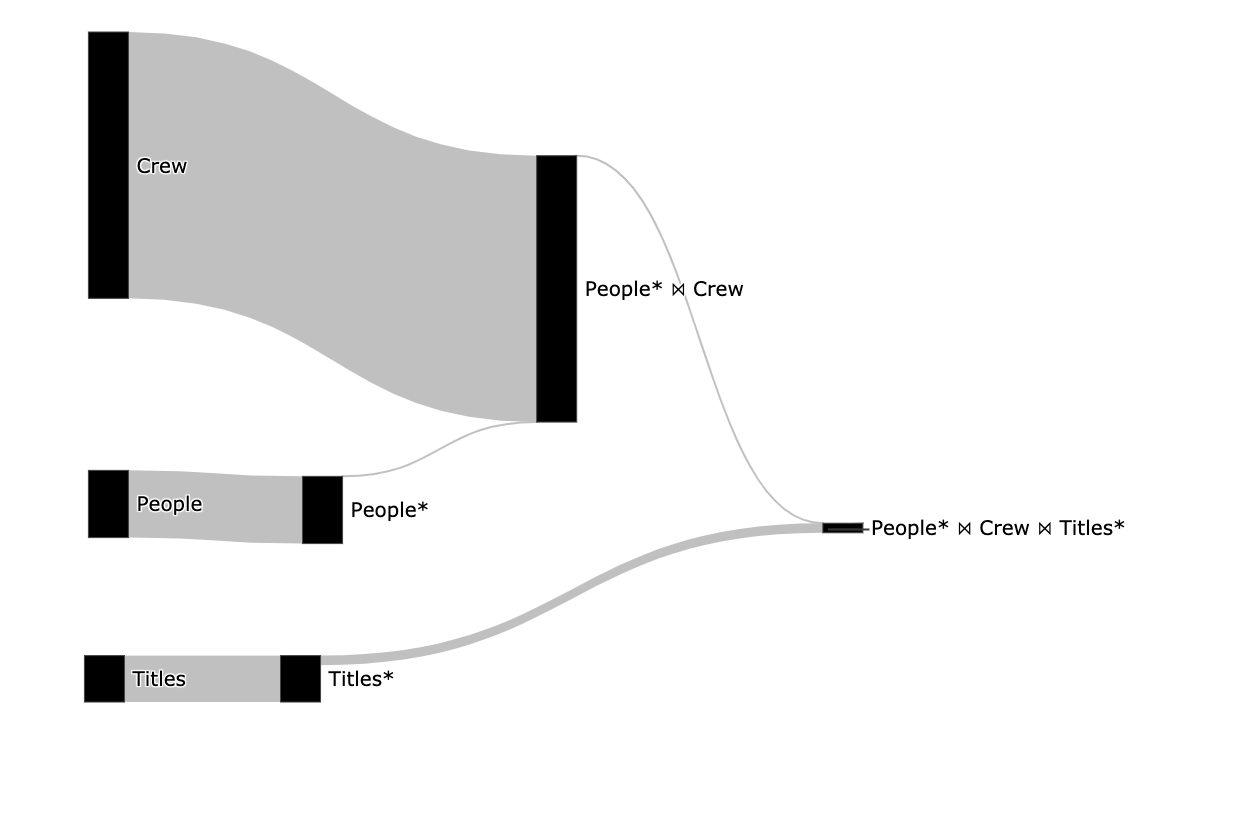
­­ **Figure 28-** Identifying missing records Sankey diagram for sub-expression   
 cardinality

Using the Sankey visualization, we can see that we got an empty result in the *People\* ⋈ Crew ⋈ Title\** relation*,* and we can see the origin of the empty result in the *Title\** relation as it is red and the *Title* link is gray.

Now that we know the problem is in the *Title* *WHERE* clause, we can fix it. The reason the predicate returns no records is that there is no lower-case *comedy* value in the *Title* relation. We will rewrite the predicate to be Camel-Case and we will support movies with multiple genres. The fixed SQL query can be seen in Figure 29.



­­ **Figure 29-** Identifying missing records fixed query  
To confirm that our query is fixed, we will use QueryFlow to visualize the cardinality again. The corresponding Sankey that represents the cardinality of our modified query can be seen in Figure 30.

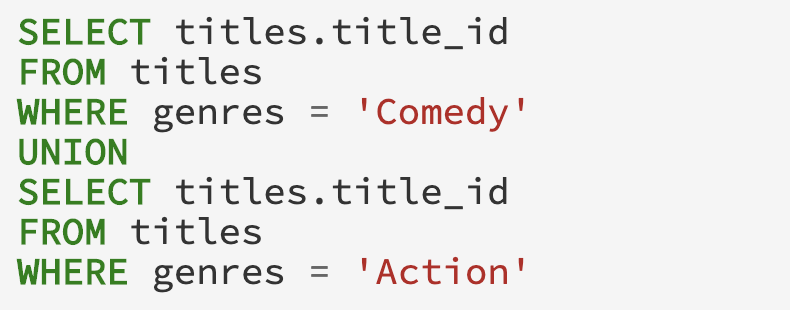


­­ **Figure 30-** Identifying missing records fixed Sankey diagram.

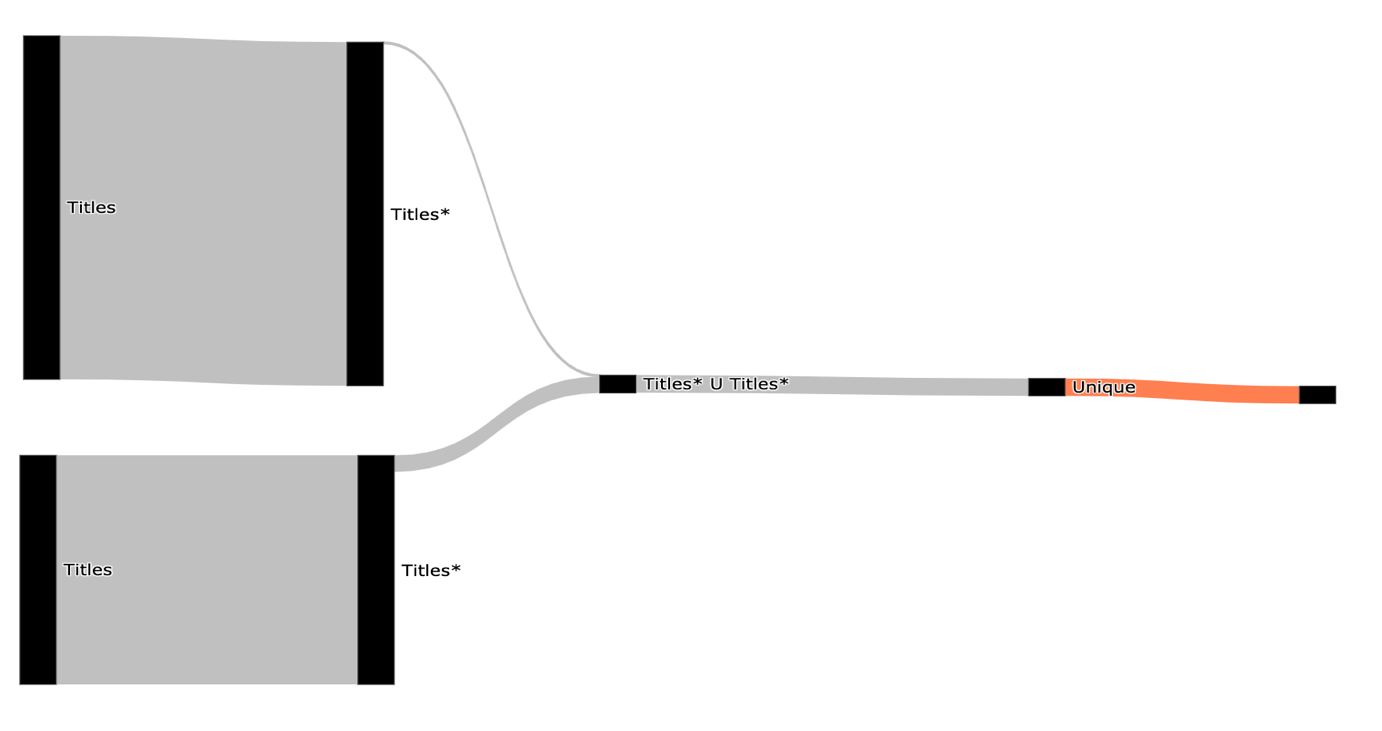
From this simple visualization in Figure 30 we can infer the following:

* We fixed the empty result set (there are no red edges).
* We understand the size of the relation, for example, *Crew* is much bigger than people and *Titles.*
* The filter on both *People* and *Titles* relations is effective and from a performance standpoint can enjoy optimization techniques like indexing and partitioning.

##### 4.2 Identifying Ineffective Operations

Problems related to infective operation are common and finding them tends to be nearly impossible with today‘s tooling. These kinds of flaws are related to performance, and although it won’t make your query wrong it will have negative effects on your DBMS’s cost, customers, and internal DBMS users as discussed in section 2.1 .   
  
Using QueryFlow we can visualize the cardinality (*actual\_rows)* of the query’ sub-expressions and find ineffective operations. Ineffective operations are operations that should change the granularity of its direct ancestor but don’t, for example, a *UNIQUE* clause that filters nothing. This problem can be caused due to either *WHERE, JOIN, UNION, HAVING,* or *UNIQUE* clauses.   
  
To show an example of an ineffective operation, we introduce the following question, “list all comedy movies and all action movies”, this question is equivalent to the SQL query in Figure 31.  
  


**Figure 31-** Identify Ineffective Operations Query  
We are going to use QueryFlow to identify the redundant sub-expression (redundant operations will have an orange edge). The corresponding Sankey that represents the cardinality of our example can be seen in Figure 32.

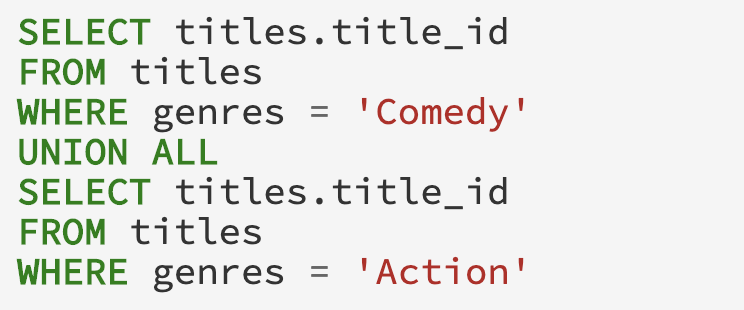


­­ **Figure 32-** Identifying ineffective operations Sankey diagram for   
 sub-expression cardinality

Using the Sankey visualization, we can see that the *Unique* sub-expression is redundant (it filters nothing) as it is marked in orange or by hovering both operations and looking at the number of rows. This affects the execution time of the query, and the query took 9.3 seconds.

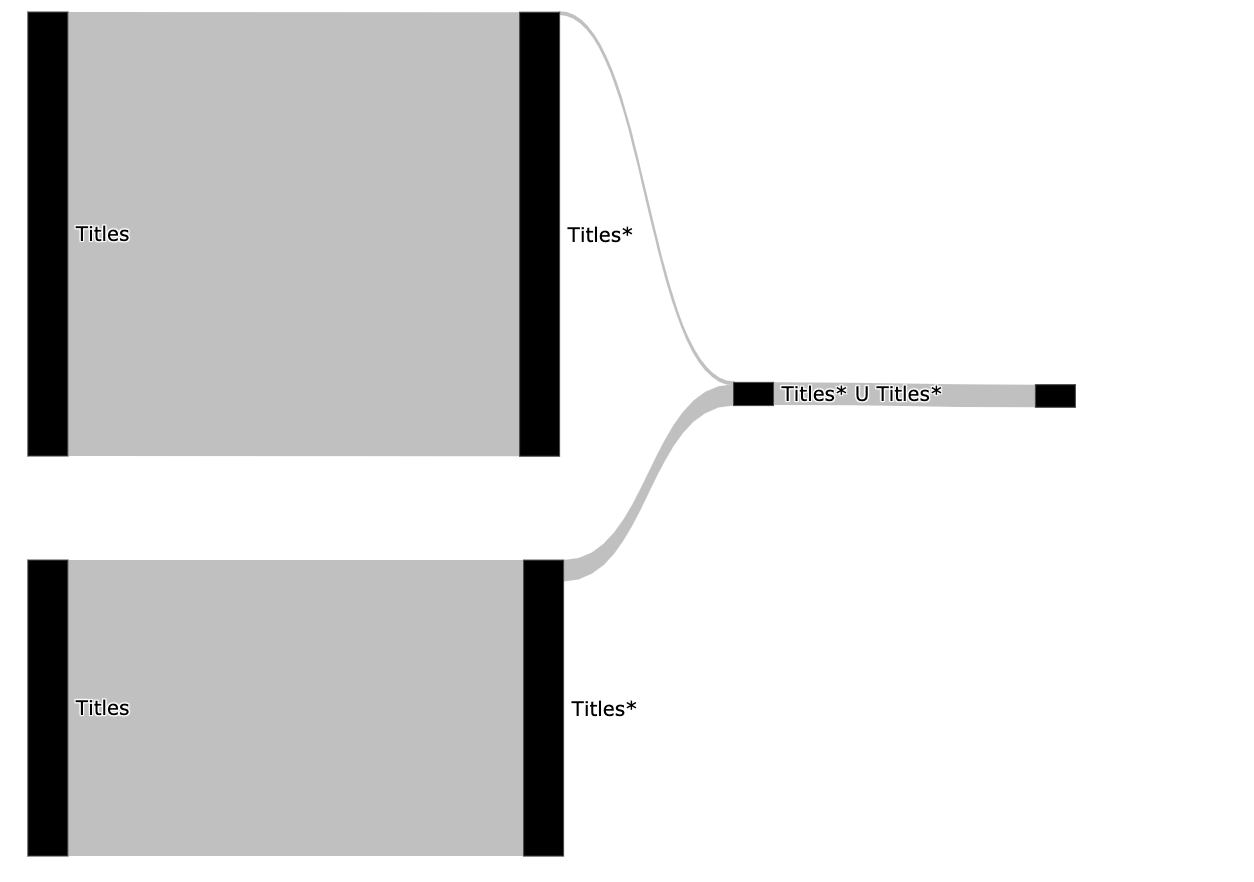
In PostgreSQL, the difference between *UNION ALL* and *UNION* is that the first only append two relations and the second removes duplicates entries after the append. Since both relations are disjoint, there is no need for removing duplications after the append. This will not affect the query correctness but will improve the query performance as *Unique* is a very expensive operation.

Now we can improve our query performance by switching the *UNION* clause with a *UNION ALL* clause. The fixed SQL query can be seen in Figure 33.



**Figure 33-** Identify Ineffective Operations Fixed Query

To confirm that our query is fixed, we will use QueryFlow to visualize the cardinality again. The corresponding Sankey that represents the cardinality of our modified query can be seen in Figure 34.



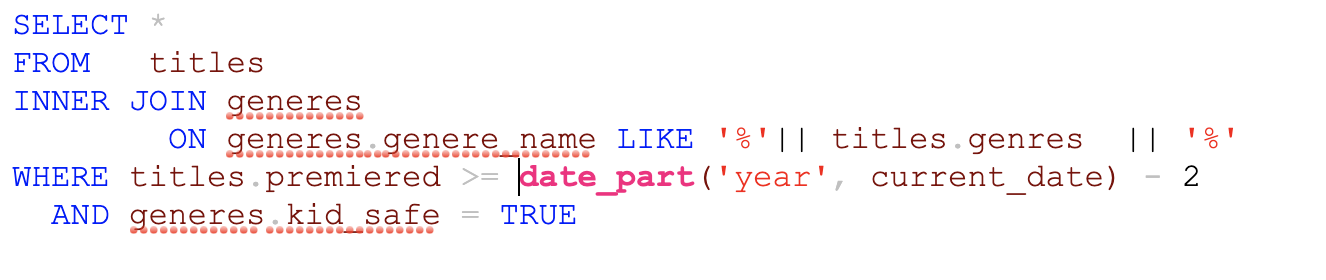
­­ **Figure 34-** QueryFlow Sankey diagram for fixed query

From this simple visualization in Figure 34 we can infer the following:

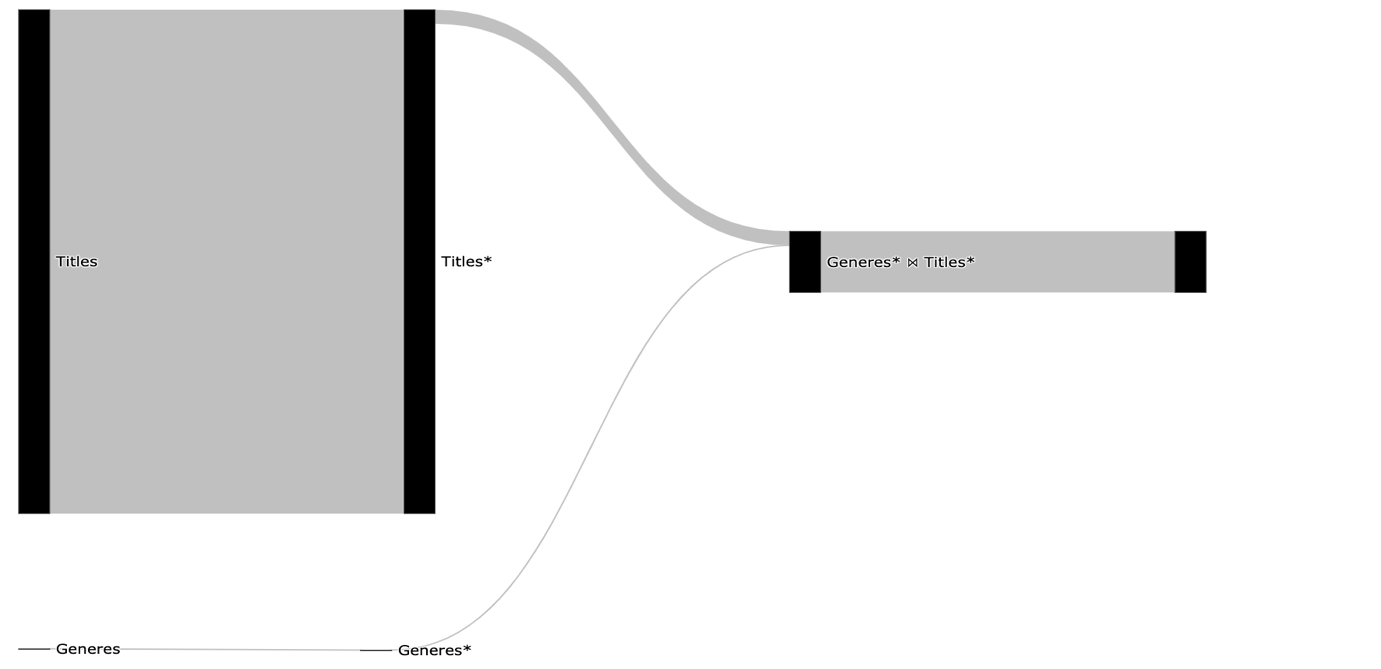
* We fixed the ineffective operation problem (there are no orange edges).
* The filter on both *People* and *Titles* relations is effective and from a performance standpoint can enjoy optimization techniques like indexing and partitioning.
* The query execution took only **1.6 seconds**, almost 6 times improvement.

##### 4.3 Identifying Duplications

Problems related to duplicates entries are common and finding them is extremely hard. By using QueryFlow to visualize the cardinality of the sub-expression of our query we can find duplications. This problem can be caused due to either *JOIN*, *UNION ALL* clauses.

To show an example of identifying duplications, we introduce the following question, “find all movies with genres that are safe for kids from the last two years”. This question is equivalent to the SQL query in Figure 35 ( the | | operation is a string concatenation in PostgreSQL).

­**Figure 35-** Identify Duplications QueryWe are going to use QueryFlow to identify duplications in our query. The corresponding Sankey that represents the cardinality of our example can be seen in Figure 36.



­­ **Figure 36-** Identify Duplications Sankey diagram

Using the Sankey visualization, we can see that *Join* sub-expression is exploding (it is bigger than its direct ancestor and we expected one to one relationship). This allows us to understand the *JOIN* condition is wrong and causes duplications.  
   
Now that we know we have a problem with the *Join* clause, we can modify the query by adding a deduplication phase. There are multiple ways to fix it, the most obvious is to add *DISTINCT* to our query. But this tends to come with big performance degradation, and we will use an equivalent query using the window function *row\_number. row\_number* give an incremented value to each element in the partition *(PARTITION BY)*, this will enable us to keep only the first occurrence of the entity we partition by. The query is the same as Figure 35 but each record with *title\_id* will auto be incremented indicator, which will help to filter duplication out. The fixed SQL query can be seen in Figure 37.

A picture containing text

Description automatically generated

**Figure 37-** Identify Duplications Fixed QueryTo confirm that our query is fixed, we will use QueryFlow to visualize the cardinality again. The corresponding Sankey that represents the cardinality of our modified query can be seen in Figure 38.

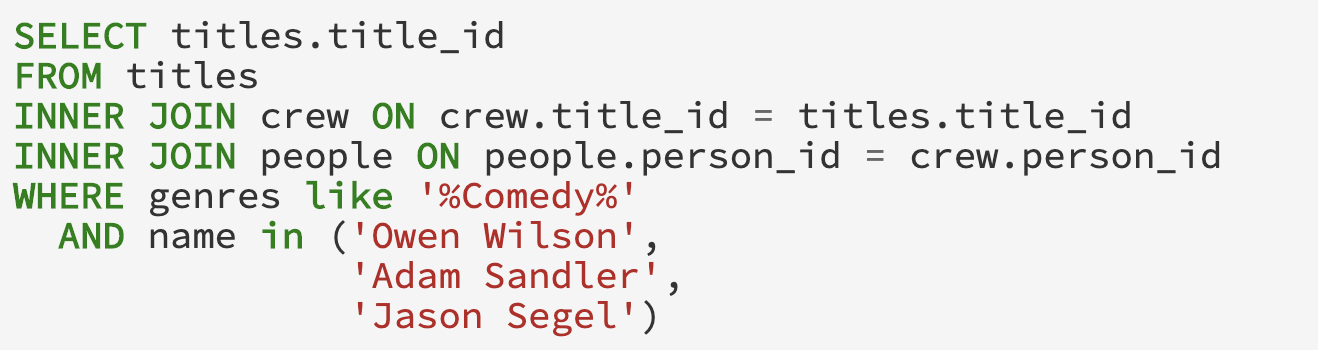
Diagram

Description automatically generated

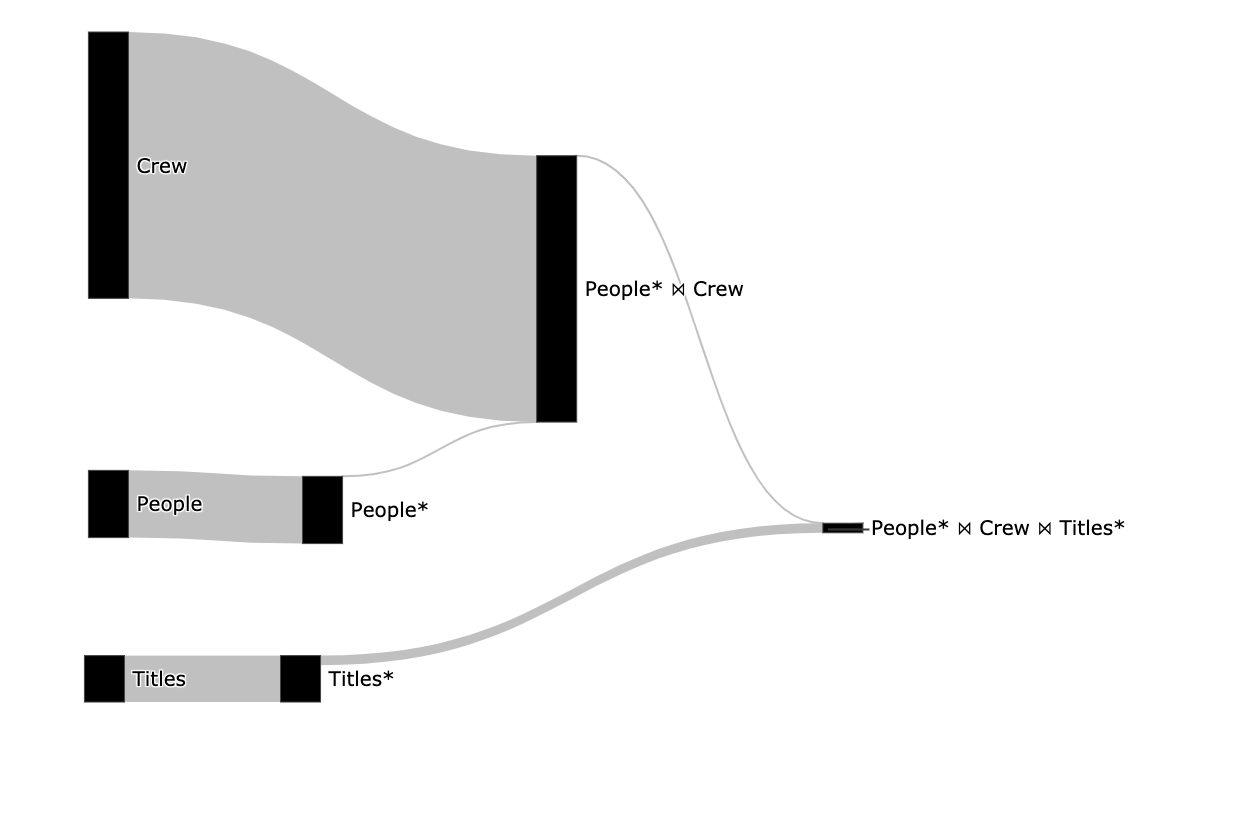
**Figure 38-** Identify Duplications Fixed Sankey diagram

We can immediately see (Figure 38) the join still introduces the duplications but after the *Sub query* clause, we are removing those duplications in the *t\** sub-expression.

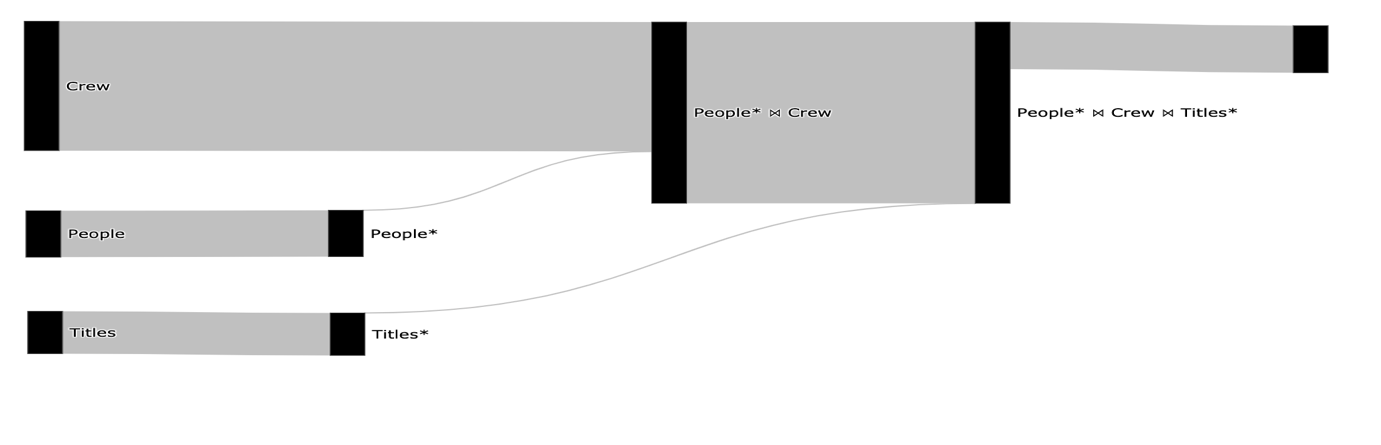
##### 4.4 Identifying Performance Bottlenecks in a Single Query

Slow queries are common and can be caused for various reasons. This makes optimizing queries extremely hard. Using QueryFlow we can visualize the query’ sub-expressions characteristics.   
  
Since the execution time of a sub-expression is affected by several factors, we might need to visualize the cardinality (*actual\_rows*), duration (*actual\_duration*), and other statistics like whether an operation spilled to disk. We will use the same example as in chapter 3, the SQL can be seen in Figure 39.  


­­ **Figure 39-** Identify Performance Bottleneck QueryWe are going to use QueryFlow to identify the bottlenecks of our query. The corresponding Sankey that represents the cardinality can be seen in Figure 40.



­­ **Figure 40-** Identify Performance Bottleneck Cardinality Sankey diagram  
We can see from Figure 40 that only a few rows are retrieved from the *People\* ⋈ Crew* due to the filter on the *People* relation, which indicates it might be a good candidate for optimization.   
To get more information, we are going to create another Sankey that represents the duration of our example, and it can be seen in Figure 41.

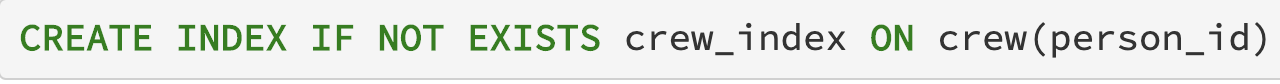


­­­­ **Figure 41-** Identify Performance Bottleneck Duration Sankey diagram

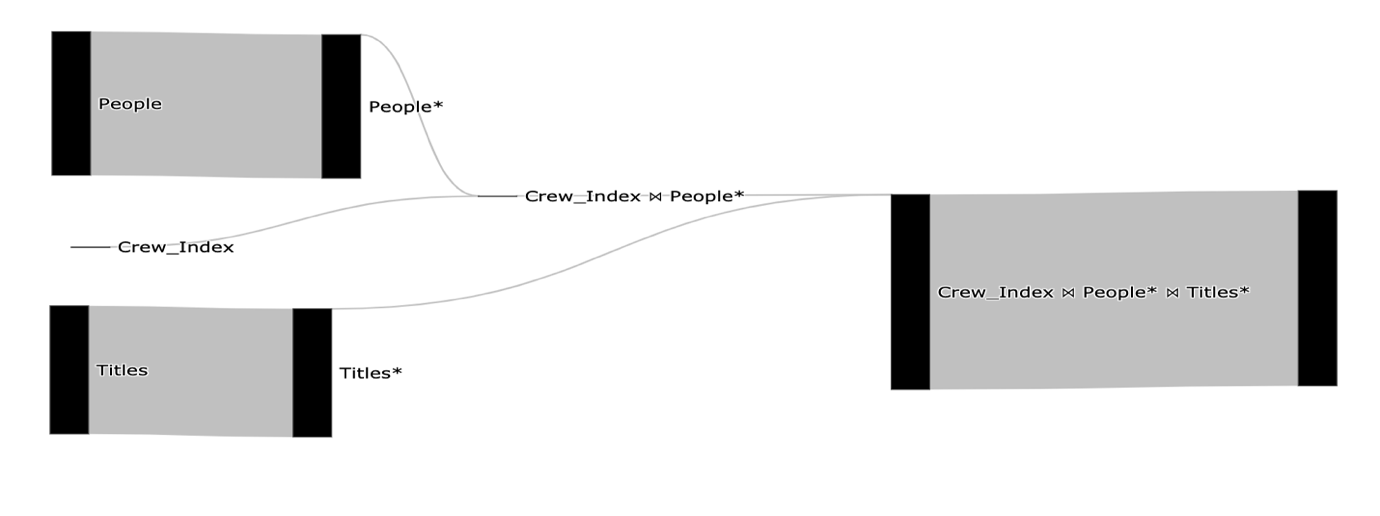
The total query duration is 7 seconds, we can see in Figure 41 that the longest operation (3.6 seconds) is the People\* ⋈ Crew.

We can improve our query performance in multiple ways. The easiest one is to add a new index on the *Crew* relation using the *person\_id* column. The reason an index will improve our query execution time is that there are a lot of rows that can be skipped in the *Crew* scan. This will allow us to use PostgreSQL’ *Hash Join* in a performant way (Index Join).

The index creation query on *person* *id* can be seen in Figure 42.



­­ **Figure 42-** Identify Performance Bottleneck Crew Index Creation  
To confirm that we eliminated the bottleneck, we will use QueryFlow to visualize the duration again. The corresponding Sankey that represents the duration of our query now can be seen in Figure 43.

****

­­ **Figure 43-** Identify Performance Bottleneck Duration Sankey diagram After Index

The total query duration is 2 seconds a 3.5 time improvement, we can see in Figure 43 that it improved both the *People\* ⋈ Crew* and the scan on the *Crew* relation.

Performance optimization is an iterative process. We can continue to improve our query in the same manner if needed.

##### 4.5 Identify flaws in the optimizer itself

Problems related to the optimizer work are hard to detect for regular users. Using QueryFlow we can visualize and compare the optimizer estimations to the actual statistics after executions.   
  
To visualize multiple metrics in the same Sankey we modify the visualization algorithm in order to adjust the luminance of the color for different metrics. The modification for the algorithm can be seen in Figure 44.

Text

Description automatically generated with medium confidence

**Figure 44-** Modifications to the visualization algorithm for multiple metrics visualizations  
  
For example, by comparing the actual and estimated cardinality (*plan\_rows, actual\_rows*) of a query’ sub-expressions we can understand if we have stale statistics.   
  
We will use the same example as in chapter 3, the query can be seen in Figure 45.

Text

Description automatically generated

**Figure 45-** Identify Flaws in the Optimizer Query  
We are going to use QueryFlow to identify if we have stale statistics and where it was originated.

The corresponding Sankey that represents the estimated cardinality compared to the actual cardinality can be seen in Figure 46.

Diagram

Description automatically generated

­­ **Figure 46-** Identify Flaws in the Optimizer Sankey diagram

We can see that we have two colors the darker gray represents the *actual\_rows* and the darker lighter grey represent the *plan\_rows*. We can see in Figure 46 that the optimizer was way off for the *Crew* scan, as the light gray edge is much thicker than the darker one.

The reason the optimizer estimation is skewed is due to PostgreSQL’s mechanism for deleting and updating records. When an update or a delete occurs, it does not create extra space in the system. PostgreSQL rather flags these tuples as *"dead tuple"* and to remove those, one needs to use the *VACUUM* clause*.*

We can clean the dead tuple in the *Crew* relation using the *vacuum* command only on the Crew relation. The *vacuum* query can be seen in Figure 47.   
  
Graphical user interface, text, application, chat or text message

Description automatically generated

**Figure 47-** Identify Flaws in the Optimizer Vacuum Command

To confirm that the optimizer statistics are up to date, we will use QueryFlow to visualize the cardinality again. The corresponding Sankey that represents the estimated cardinality compared to the actual cardinality after the *vacuum* command can be seen in Figure 48.

**Diagram

Description automatically generated**

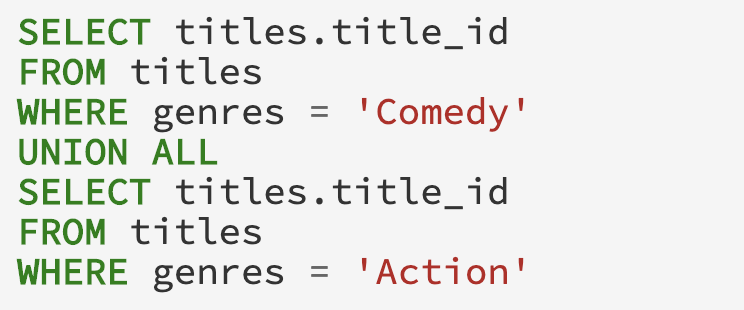
­­ **Figure 48-** Identify Flaws in the Optimizer Fixed Sankey diagram

We can immediately see in Figure 48 that the *Crew* scan is no longer skewed as used to be, as the darker and lighter edges of the *Crew* sub-expression are pretty proportional.

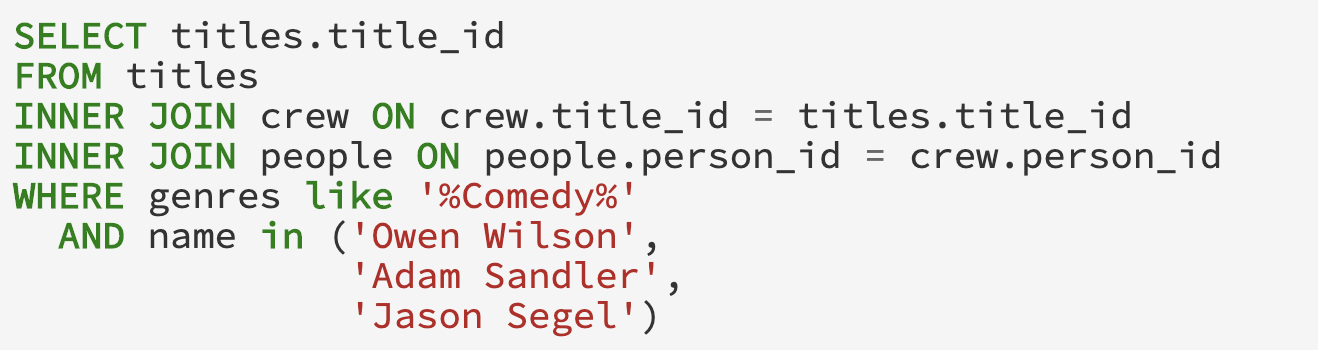
4.6 Identifying Performance Bottlenecks in Multiple Queries   
  
Slow queries are common and can be caused for various reasons. This makes optimizing queries extremely hard. The main reason multi queries optimization is even harder is that one query may affect the other. Thus, it is critical to make sure that take other queries into consideration.   
  
Using QueryFlow we can visualize the queries’ sub-expressions characteristics. Since the execution time of a sub-expression can be caused by several reasons, we might need to visualize the cardinality (actual\_rows), duration (actual\_duration), and other statistics like whether the operation spilled to disk.  
  
We can use QueryFlow as is and visualize two separate Sankies for each our queries. But we can make it even easier if we will construct one Sankey for multiple queries. To do so we will need to modify both the parsing and the visualization algorithm.   
  
In the parsing algorithm, we will add a step that checks if a sub-expression was already seen and assign the current sub-expression with that expression id. This will allow us to see the same compact our sub-expressions “share” nodes. The modification for the algorithm can be seen in Figure 49.  
  
Figure 49: Modifications to the parsing algorithm for multiple queries

To make differentiating the two queries, we will assign each query with its color. The modification for the visualization algorithm can be seen in Figure 50.

Figure 50: Modifications to the visualization algorithm for multiple

To illustrate the gist of the MQO problem, we use two simple queries. The first query is the same example as in 4.2. The query can be seen in Figure 51. 

­­ **Figure 51 -** Identify Performance Bottleneck First Query  
The second query we will use is the same example as in chapter 3. The query can be seen in Figure 52.



**Figure 52-** Identify Performance Bottleneck Second Query

We are going to use QueryFlow to identify the bottlenecks of our queries. To do so we are going to visualize both of our queries in one Sankey-diagram. This will enable us to represent logically similar operations with the same node. For example, since both of the queries use the *Title* relation and have a filter on *comedy,* they will share both the *Title* node and the *Title* node that represent the *comedy* filter (as can be seen with red rectangle). The corresponding Sankey that represents the cardinality of our example can be seen in Figure 53.

Diagram

Description automatically generated

**Figure 53-** Identify Performance Bottleneck Multiple Queries   
 Cardinality Sankey diagramWe can see from Figure 53 that only a few rows are retrieved from the *People\* ⋈ Crew* due to the filter on the *People* relation and the same goes for filter *Title* relation, which indicates there might be a good candidate for optimization.   
To get more information, we are going to create another Sankey that represents the duration of our example, and it can be seen in Figure 54.

A picture containing timeline

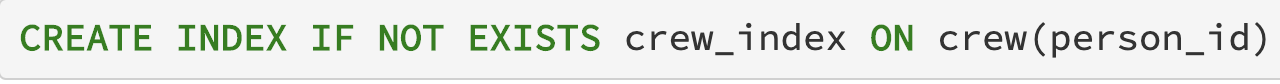
Description automatically generated­­

**Figure 54** Identify Performance Bottleneck Multiple Queries   
 Duration Sankey diagram

The total query duration of both queries is 7.5 seconds, we can see in Figure 54 that the longest operation (3.6 seconds) is the People\* ⋈ Crew.

We can improve our query performance in multiple ways. The easiest one is to add a new index on the *Crew* relation using the *person\_id* column. The reason an index will improve our query execution time is that there are a lot of rows that can be skipped in the *Crew* scan. This will allow us to use PostgreSQL’ *Hash Join* in a performant way (Index Join).

The index creation query on the crew relation with the person\_id column can be seen in Figure 55.



­­ **Figure 55-** Identify Performance Bottleneck Crew Index Creation

To confirm that we eliminated the bottleneck, we will use QueryFlow to visualize the duration again. The corresponding Sankey that represents the duration of our query now can be seen in Figure 56.

**Diagram

Description automatically generated**

­­ **Figure 56-** Identify Performance Bottleneck Duration Sankey diagram   
 After Crew Index Creation

The total query duration is 4.5 seconds, we can see in Figure 56 that it improved both the *People\* ⋈ Crew* and the scan on the *Crew* relation.

Performance optimization is an iterative process. To illustrate the MQO in the best way, we will continue to improve our queries. We see that both queries uses*Title* scan and in particular have predicates. We will add an index on the Titles relation using the title\_id and genres columns. The index creation query on the titles relation can be seen in Figure 57.

  
­­ **Figure 57-** Identify Performance Bottleneck Title Index Creation  
To confirm that we eliminated the bottleneck, we will use QueryFlow to visualize the duration again. The corresponding Sankey that represents the duration of our query now can be seen in Figure 58.

A picture containing diagram

Description automatically generated

­­ **Figure 58-** Identify Performance Bottleneck Duration Sankey diagram   
 After Title Index Creation

The total query duration is 3.5 seconds, we can see in Figure 58 that it improved both of the table Scans on the Titles\* relation. We can continue to improve our query in the same manner if needed.

It’s important to understand that it is not practical to take every pair of queries and do the same process. In chapter 5 we see an example of how to use QueryFlow on 20+ queries effectively.

##### 4.7 When QueryFlow won’t help

The chapter covered use cases and flaws QueryFlow can help identify. But, like all tools, QueryFlow won’t be effective in all the use cases.  
  
QueryFlow visualizes query sub-expressions according to a measurable metric. Thus, if we have very skewed values, some of the insights that can be inferred will be less visible.   
  
For example, in Figure 59 we the edge value between the “Team A” node to the “Sold” node is 9 times bigger than the edge between the “Team A” node to the “Not Sold” but since the other relations are much smaller it will skew the visualization.  
  
A picture containing diagram

Description automatically generated

**­­ Figure 59-** Skew problems  
  
To mitigate it, we can represent the same metric using percentage, this will reduce the skewness and the visualization will look closer to reality as can be seen in Figure 60/  
A picture containing diagram

Description automatically generated

­­ **Figure 60-** Skew problems mitigation  
  
In addition, when we have a very complex Sankey-diagram one can be overwhelmed with information. In Figure 61 I show an example of 100 teams with 25 sales phases, and we can see that it’s hard to make sense which ones are bigger.

A screenshot of a computer

Description automatically generated with low confidence

­­ **Figure 61-** complexity problem

This can be mitigated by filtering and visualizing only promising candidates, and in chapter 5 I will show an example on TPC-H.

Another limitation of QueryFlow is that it relies on the execution plan information. If the root cause can’t be identified using any of the statistics from the execution plan, QueryFlow won’t be able to help either.

# Chapter 5:Evaluation

The goal of this evaluation is to objectively quantify the impact and benefits that can be achieved using QueryFlow for performance optimization. To evaluate our solution, I’m going to use the TPC-H benchmark queries (22) on PostgreSQL.

The rest of this chapter is structured as follows:

* Section 5.1 provides an overview of TPC-H
* Section 5.2 provides an overview of the evaluation setup.
* Section 5.2 evaluate optimizations for scale factor 1.
* Section 5.3 evaluate optimizations for scale factor 10.
* Section 5.4 provides an overview of lessons that can be learned for SQL best practices.

##### 5.1 TPC-H

To evaluate our solution, I’m going to use the TPC-H benchmark on PostgreSQL. TPC-H benchmark as a tool to help database vendors develop their database engines. TPC-H consists of a suite of business-oriented ad-hoc queries and concurrent data modifications, that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

TPC-H dataset contains a total of eight tables (*region, nation, part, supplier, partsupp, customer, orders,* and *lineitem*). The tables and their relations can be described in Figure 62.  
   
Diagram

Description automatically generated  
­­. **Figure 62-** TPC-H Schema

The reason I picked TPC-H is that it’s well understood in academics, and the queries and the data have been chosen to have broad industry-wide relevance and they are divided into the following groups:

|  |  |  |
| --- | --- | --- |
| **Group** | **Features** | **Queries** |
| A | * Medium dimensionality * Result is TPC-H scale factor independent | Q1, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q12, Q13, Q14, Q16, Q19, Q22 |
| B | * High dimensionality * Few results, lots of empty cells | Q15, Q18 |
| C | * High dimensionality * Result is TPC-H scale factor dependent | Q2, Q9, Q10, Q11, Q17, Q20, Q21 |

In addition, TPC-H is very strict in nature to make a fair engine comparison. This is understandable, as allowing tricks like materialized views would make it trivial to tune the workload. Some of TPC-H limitation are:

* You may index a primary key.
* You may index a foreign key.
* You may partition any table on one and only one column that has the type date.
* This partitioning can be done down to the day level.
* More

When evaluating TPC-H results, it is important to understand the concept of Scale Factor (SF). The scale factor measures the size of the input data. To load TPC-H data, we are going to use dbgen to generate CSV files representing our tables and we then will load them.

##### 5.2 Setup

All the experiments were performed on a single machine with 32 GB memory and 8 cores, and the optimization times are measured as CPU time (user + system).   
  
We executed each benchmark several times for each scale factor to provide a more comprehensive evaluation. We executed the TPC-H benchmark with the following scales:

* Scale 1 - Consists of the base row size (several million elements) and is 1 GB in size.
* Scale 10 - Consists of the base row size x 10 and is 10 GB in size.

For indices, we are going to use Dexter, an open-source index selection tool for PostgreSQL [21, 22]. Dexter identifies and create suitable index combinations by analyzing the EXPLAIN command and how index creation will affect it. For all scales Dexter decided on the following optimizations:

* Indices on *lineitem* relation:
  + *CREATE INDEX ON "lineitem" ("l\_orderkey")*
  + *CREATE INDEX ON "lineitem" ("l\_partkey")*
  + *CREATE INDEX ON "lineitem" ("l\_shipmode", "l\_partkey")*
* Indices on *partsupp* relation:
  + *CREATE INDEX ON "partsupp" ("ps\_partkey")*
  + *CREATE INDEX ON "partsupp" ("ps\_suppkey")*

As we can it only suggests indices on the facts, and as we can see those indices are pretty vanilla obvious (on the facts, no partial indices or ordered indices).

For the Scale 1 ~~/10~~ GB benchmark, we can benchmark average is around 22/~~TODO~~ seconds, and the query distribution can be seen in Figure 62. explain histogram and give the query that was improved. Add number of queries and the duration axies  
 Chart, histogram

Description automatically generated **Figure 62-** TPC-H SF-1 Baseline Histogram

##### 5.2 Evaluate optimizations for scale factor 1 As we described TPC-H consists of a suite of business-oriented ad-hoc queries with a high degree of complexity. When we are optimizing multiple complex queries, one query may affect the other and tend to be out of the optimizer scope. We will use QueryFlow to identify the bottlenecks across TPC-H’s queries to reduce the execution time. The execution time Sankey of the 22 queries can be seen in Figure 59, each query will have different color and the edge width is proportional to the execution time.

Graphical user interface, application

Description automatically generated

­­

**Figure 63-** TPC-H SF-1 Sankey-diagram for all queries

Figure 63 can be overwhelming, as we have a lot of nodes and edges. We can see that there few long queries, that take most of the execution time. Thus, we will benefit from checking only the heaviest queries, and after we finished optimizing, if further optimization is required, we can re-iterate.   
  
We will visualize only the 7 heaviest as can be seen in Figure 64.

Timeline

Description automatically generated

**Figure 64-** TPC-H SF-1 Sankey-diagram for slowest queries

Now it much easier to understand the heaviest parts of the queries. The first thing we would like to check is whether it’s A systematic issue like database configurations as we heavy joins and aggregations.

The immediate candidate is a low *work\_mem* configuration, but, thanks to the additional information QueryFlow shows when we over we can see there were no spills to disk, and we are ok.

We will start by doing the following optimizations to the database:

* Indices on orders relation:
  + CREATE INDEX ON "orders" ("o\_custkey")
* Partitioned both *ORDER* and *LINEITEM* on date

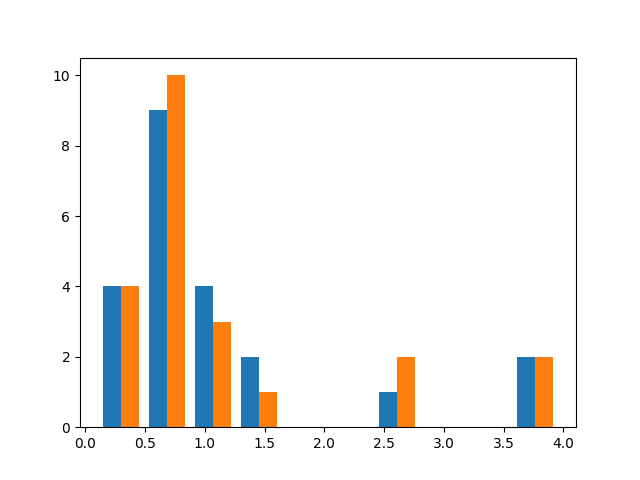
As a reminder before the optimization TPC-H took 22 seconds and the histogram can be seen in Figure 62, after the optimization we got to 14 seconds and you can see the histogram in Figure 65

Chart, histogram

Description automatically generated

**Figure 65-** TPC-H SF-1 Optimized Histogram

A comparison between the baseline and the optimization can be seen in Figure 62 explain histogram and give the query that was improved. Add number of queries and the duration axies



**Figure 62-** TPC-H SF-1 Optimized vs Baseline

##### ~~5.3 Evaluate optimizations for scale factor 10~~

##### ~~We will use QueryFlow to identify the bottlenecks across TPC-H’s queries to reduce the execution time. The execution time Sankey of the 22 queries can be seen in Figure 61, each query will have different color and the edge width is proportional to the execution time.~~

**Figure 63**

~~DB Configurations~~

* + ~~shared\_buffer/effective\_cache\_size =10GB~~
  + ~~work\_mem = 1GB~~
  + ~~max\_wal\_size = 24GB~~
  + ~~max\_parallel\_workers\_per\_gather = 16~~
  + ~~max\_parallel\_maintenance\_workers = 16~~
  + ~~max\_worker\_processes = 32~~
  + ~~max\_parallel\_workers = 32~~

~~psql tpch -c "SELECT name, setting, unit, min\_val, max\_val, context FROM pg\_settings WHERE name in ('work\_mem','shared\_buffers','effective\_cache\_size','shared\_buffers', 'max\_parallel\_workers', 'max\_worker\_processes', 'max\_parallel\_workers\_per\_gather')"~~

~~As a reminder before the optimization TPC-H took 22 seconds and the histogram can be seen in Figure 58, after the optimization we got to 14 seconds and you can see the histogram in Figure 64~~

**Figure 64**

##### ~~5.4 Lesson Learned for SQL Best Practices~~

* ~~Large joins~~
* ~~Flattening subqueries~~
* ~~Rewrite like~~
* ~~Moving predicate to subquery~~
* ~~Configuration matter~~
* ~~Ordered indices~~
* ~~The schema is normalized but the queries are more olap~~
* ~~Late Projection~~
* ~~Aggregate then join~~

# Conclusions and Future Directions

In this thesis, we present our novel method for identifying flaws in SQL queries. Our method allows identifying both errors and bottlenecks of SQL queries. To facilitate this task, we have described an approach that can automatically transform SQL queries’ execution plans into Sankey-diagrams. This gives the users an intuitive understanding of the query characteristics by observing how the query is executed under the hood.   
  
To the best of our knowledge, our work is the first work that utilizes Sankey-diagrams to visualize SQL queries’ characteristics and the first work to visualize multiple SQL queries compactly.

The main advantage of our method over existing approaches is its high applicability – it can be applied to multiple queries, to identify different types of flaws without modifying the database itself and both on the logical execution time and actual execution time.   
  
We demonstrate its applicability through a proof-of-concept implementation. While using PostgreSQL as the database for this work, we have implemented our solution solely on the execution plan of PostgreSQL and conducted an experimental evaluation. The experimental results show that QueryFlow can assist in optimizing bottlenecks in a matter of minutes.

The challenge of identifying flaws in SQL queries is far from being solved. For future work, we plan to enrich the execution plan with information from the database internal tables and configurations to allow fixing additional flaws and heuristics with QueryFlow, for example, if we have spilled to disk. In addition, for huge and complex queries QueryFlow output can be overwhelming, we plan to address these issues by introducing another layer that enables the filtering of unreverent sub-expressions or queries.

# References

[1] https://en.wikipedia.org/wiki/Sankey\_diagram

[2] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu.   
 Making database systems usable. In SIGMOD, 2007.

[3] A. Deshpande, Z. Ives, and V. Raman, “Adaptive query processing,” Found. Trends   
 databases, vol. 1, no. 1, pp. 1–140, Jan. 2007. [Online]. Available:   
 http://dx.doi.org/10.1561/1900000001

[4] Y. E. Ioannidis, “Query optimization,” ACM Comput. Surv., vol. 28, no. 1, pp. 121–123,   
 Mar. 1996. [Online]. Available: <http://doi.acm.org/10.1145/234313.234367>

[5] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and   
 Yannis Velegrakis. 2014. IQR: An Interactive Query Relaxation System for the Empty-  
 answer Problem. In Proceedings of the 2014 ACM SIGMOD International Conference on

Management of Data (SIGMOD ’14).   
[6] Bidoit, N., Herschel, M., Tzompanaki, K.: Query-based why-not provenance with   
 NedExplain. In: proceeding of Int. Conference on Extending Database Technology   
 (EDBT), pp. 145–156 (2014).

[7] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2016. QFix: Demonstrating Error   
 Diagnosis in Query Histories. In Proceedings of the 2016 International Conference on   
 Management of Data (SIGMOD ’16). San Francisco, California, USA.

[8] A[R. Caballero, Y. Garc´ıa-Ruiz, and F. S´aenz-P´erez. Algorithmic Debugging of SQL   
 Views. In proceeding of Ershov Informatics Conference (PSI’11), Lecture Notes in   
 Computer Science. Springer, 2011. In Press.](https://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-3-11.pdf)

[9] [B. Dietrich and T. Grust. A SQL Debugger Built from Spare Parts—Turning a SQL:1999   
 Database System into its Own Debugger. In Proceeding of ACM SIGMOD, Melbourne,   
 Australia, 2015.](https://db.inf.uni-tuebingen.de/staticfiles/publications/debugger-spare-parts.pdf)

[10] [J. Danaparamita and W. Gatterbauer. QueryViz: helping users understand SQL queries and   
 their patterns. In proceeding of EDBT/ICDT ’11, Uppsala, Sweden, 2011.](https://openproceedings.org/2011/conf/edbt/DanaparamitaG11.pdf)

[11] [E. A. Silva, N. M. Franco, Ma.o R. Ferro and R. N. Fidalgo. Mental Workload Impact of a   
 Visual Language on Understanding SQL Queries](https://www.br-ie.org/pub/index.php/sbie/article/download/8728/6289), 2019. In Brazilian Symposium on   
 Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE), vol. 30,   
 no. 1, p. 239. 2019.

[12] D. Moritz et al. Perfopticon: Visual query analysis for distributed databases. Computer   
 Graphics Forum (Proc. EuroVis), 2015.

[13] [J. R. Haritsa. The Picasso database query optimizer visualizer. PVLDB, 3(2):1517–1520,   
 2010.](http://vldbarc.org/pvldb/vldb2010/pvldb_vol3/D01.pdf)

[14] Sneha Gathani, Peter Lim, and Leilani Battle. 2020. Debugging Database Queries: A   
 Survey of Tools, Techniques, and Users. In Proceedings of the 2020 CHI Conference on   
 Human Factors in Computing Systems (CHI '20). Association for Computing Machinery,   
 New York, NY, USA, 1–16. DOI:https://doi.org/10.1145/3313831.3376485

[15] Baryamureeba, Venansius & Ngubiri, John. (2004). On Improvement of the Volcano   
 Search and Optimization Strategy. 839-846. 10.1007/11558958\_101.

[16] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan   
 Demers. 2009. Rule-based multi-query optimization. In Proceedings of the 12th   
 International Conference on Extending Database Technology: Advances in Database   
 Technology (EDBT '09). Association for Computing Machinery, New York, NY, USA,   
 120–131. DOI:https://doi.org/10.1145/1516360.1516376

[17] Havre, S., Hetzler, E.G., Perrine, K.A., Jurrus, E., & Miller, N. (2001). Interactive   
 visualization of multiple query results. IEEE Symposium on Information Visualization,   
 2001. INFOVIS 2001., 105-112.

[18] Paoli, Leonardo & Lupton, Richard & Cullen, Jonathan. (2017). Probabilistic model   
 allocating primary energy to end-use devices. Energy Procedia. 142. 2441-2447.   
 10.1016/j.egypro.2017.12.180.

[19] linear programming

[20] Sara Di Bartolomeo, Yixuan Zhang, Fangfang Sheng, Cody Dunne, "Sequence Braiding:   
 Visual Overviews of Temporal Event Sequences and Attributes", Visualization and   
 Computer Graphics IEEE Transactions on, vol. 27, no. 2, pp. 1353-1363, 2021

[21] M.Y.L.Ip, L.V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes.   
 IEEE Transactions on Software Engineering, 9(2):135–143, 1983.

[22] A. Kane. Dexter - The automatic indexer for Postgres, June 2017.   
 <https://github.com/ankane/dexter>, visited 2020-06-04.