**The Open University of Israel**

**Department of Mathematics and Computer Science**

**Thesis Proposal**

**Visualizing Database Execution Plans using Sankey**

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

2020

# Acknowledgements

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 users’ queries or in the database configuration.

Databases are complex systems, and often lack the tooling to identify flaws in your queries and their origins, such as debuggers and profilers. Without proper tooling, it is rare for users to write flawless queries in terms of errors or any performance implications.

In our work, we focus on identifying cardinality problems and performance bottlenecks. Our goal is to give proper tooling to identify flaws in SQL queries, that can be easily integrated with the existing real-world systems. Therefore, we assume that the solution does not modify the databases themselves.

We have implemented QueryFlow, a query visualization tool that provides insights into characteristics of the query and help identify common DBMS’ problems, such as performance bottlenecks and cardinality issues. QueryFlow can be integrated with various databases and onboarding a new one, require only to implement a parser for the execution plan.

The experimental results show that our solution allows us to pinpoint and fix the flaws in our 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: 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 2.4: Debugging Approach ………………………………………… 16  
Section 2.5: Query Visualization Approach …………………………... …..17  
Section 2.6: Approaches Comparison …………………………………….. 20  
Section 2.7: Multi Query Optimization …………………………………… 21  
Section 2.8: Sankey Diagrams …………………………………………….. 22  
Section 2.9: 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  
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: Identifying Performance Bottlenecks in Multiple Queries …… 54  
Section 4.6: Identify flaws in the optimizer itself ………………………….. 58  
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: Overview of IQR ………………………………………………………………………………….….. 15

Figure 2: Overview of QFix …………………………………………………………………………………….. 16

Figure 3: Habitat query's breakpoint interface ………………………………………………………………….. 17

Figure 4: Habitat query sub-expression view …………………………………………………………………… 17

Figure 5: Overview of QueryViz ……………………………………………………………………………….. 18

Figure 6: Overview of Perfopticon ……………………………………………………………………………… 19

Figure 7: Overview of Picaso …………………………………………………………………………………… 19

Figure 8: Sales Teams Sankey-diagram ………………………………………………………………………… 22

Figure 9: Sales Teams Sankey-diagram hovered ……………………………………………………………….. 23

Figure 10: Energy ecosystem Sankey-diagram ……………………………………………………………......... 23

Figure 11: Cancer research Sankey-diagram ……………………………………………………………………. 24

Figure 12: QueryFlow Design …………………………………………………………………………………. . 28

Figure 13: QueryFlow Enrichment algorithm…………………………………………………………………… 30

Figure 14: IMDB Schema ……………………………………………………………………………………….32

Figure 15: Detailed Example – Query …………………………………………………………………………...33

Figure 16: Detailed Example - Explain Query …………………………………………………………………..33

Figure 17: Detailed Example - Explain Analyze Query …………………………………………………………34

Figure 18: Detailed Example - Explain Analyze Results ………………………………………………………..34

Figure 19: Detailed Example - Explain Results …………………………………………………………………35

Figure 20: Detailed Example - Tree representation of execution plan …………………………………………..36

Figure 21: Detailed Example - Tree representation of execution plan after QueryFlow parsing phase …………37

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

Figure 23: Detailed Example - Table representation of execution plan for QueryFlow visualization phase ……41

Figure 24: Detailed Example - QueryFlow Sankey diagram for sub-expression cardinality ……………………42

Figure 25: Identify Missing Records – Query …………………………………………………………………...45

Figure 26: Identify Missing Records - QueryFlow Sankey diagram for query's sub-expression cardinality ……45

Figure 27: Identify Missing Records - Fixed Query ……………………………………………………………..46

Figure 28: Identify Missing Records - QueryFlow Sankey diagram for fixed query's   
 sub-expression cardinality ……………………………………………………………………………46

Figure 29: Identify Ineffective Operations – Query ………...……………………………………………………47

Figure 30: Identify Ineffective Operations - QueryFlow Sankey diagram for query's   
 sub-expression cardinality …………………………………………………………………………… 48

Figure 31: Identify Ineffective Operations - Fixed Query ………………………………………………………. 48

Figure 32: Identify Ineffective Operations - QueryFlow Sankey diagram for fixed query's   
 sub-expression cardinality …………………………………………………………………………….49

Figure 33: Identify Duplications – Query ……………………………………………………………………….. 50

Figure 34: Identify Duplications - QueryFlow Sankey diagram for query's sub-expression cardinality ……….. 50

Figure 35: Identify Duplications - Fixed Query ………………………………………………………………… 51

Figure 36: Identify Duplications - QueryFlow Sankey diagram for fixed query's sub-expression cardinality …. 51

Figure 37:Identify Performance Bottleneck in a Single Query – Query …………………………………………52

Figure 38: Identify Performance Bottleneck in a Single Query - QueryFlow Sankey diagram   
 for query's sub-expression cardinality ………………………………………………………………. 52

Figure 39: Identify Performance Bottleneck in a Single Query - QueryFlow Sankey diagram   
 for query's sub-expression duration ………………………………………………………………….. 53

Figure 40: Identify Performance Bottleneck in a Single Query - Index creation………………………………. 53

Figure 41: Identify Performance Bottleneck in a Single Query - QueryFlow Sankey diagram   
 for query's sub-expression duration after optimization …………………………………………….. 54

Figure 42: Identify Performance Bottleneck in Multiple Queries - Query 1 ……………………………………54

Figure 43: Identify Performance Bottleneck in Multiple Queries - Query 2 ……………………………………55

Figure 44: Identify Performance Bottleneck in Multiple Queries - QueryFlow Sankey diagram   
 for query's sub-expression cardinality ……………………………………………………………….55

Figure 45: Identify Performance Bottleneck in Multiple Queries - QueryFlow Sankey diagram   
 for query's sub-expression duration ………………………………………………………………….56

Figure 46: Identify Performance Bottleneck in Multiple Queries - Crew Index creation ……………………….56

Figure 47: Identify Performance Bottleneck in Multiple Queries - QueryFlow Sankey diagram   
 for query's sub-expression duration with Crew index ……………………………………………….57

Figure 48: Identify Performance Bottleneck in Multiple Queries - Titles Index creation ………………………57

Figure 49: Identify Performance Bottleneck in Multiple Queries - QueryFlow Sankey diagram   
 for query's sub-expression duration with Title index ………………………………………………..58

Figure 50: Identify Flaws in the Optimizer – Query ……………………………………………………………58

Figure 51: Identify Flaws in the Optimizer - QueryFlow Sankey diagram   
 for query's sub-expression cardinality and estimated cardinality ……………………………………59

Figure 52: Identify Flaws in the Optimizer – Vacuum ………………………………………………………….59

Figure 53: Identify Flaws in the Optimizer - QueryFlow Sankey diagram for query's

sub-expression cardinality and estimated cardinality after vacuum …………………………………60

Figure 54: When QueryFlow won’t help - Skew problems ……………………………………………………..60

Figure 55: When QueryFlow won’t help - Skew problems mitigation ………………………………………….61

Figure 56:When QueryFlow won’t help - complexity problem …………………………………………………61

Figure 57: TPC-H Schema ………………………………………………………………………………………64

Figure 58: TPC-H SF-1 Baseline Histogram...…………………………………………………………………...64

Figure 59: TPC-H SF-1 Sankey-diagram for all queries …………………………………………………..…… 65

Figure 60: TPC-H SF-1 Sankey-diagram for slowest queries ………………………………………………….. 65

Figure 61: TPC-H SF-1 Optimized Histogram …………….. ………………………………………………….. 66

Figure 62: TPC-H SF-1 Optimized vs Baseline ……………..………………………………………………….. 66

Figure 63: TPC-H SF-1 Sankey-diagram for slowest queries ………………………………………………….. 67

Figure 63: 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, since SQL queries tend to be verbose and involve complex logic, non-trivial queries are hard to perfect, even for SQL experts. Hence, the ability to understand and “debug“ the execution of a SQL query is a necessary step towards using DBMSs effectively.  
  
Identifying those flaws in complex queries and bringing those queries to perfection is challenging, and often requires more than fixing syntax issues. SQL queries can return zero entries, duplicate entries, unexpected results, or not meet the minimal performance requirements. To mitigate it, most databases provide detailed statistics of the intermediate sub-expressions about the query that is about the be executed. But, due to SQL’s declarative nature, the translation from a query sub-expression to its corresponding part in the execution plan is difficult for most users.

##### 1.2 Motivation

To help users find and fix their queries in a more intuitive way, techniques like execution plan visualizations were invented. These techniques give the users a much more intuitive understanding of their query, by observing how the query is executed (or planned) under the hood and its 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. Our main contributions in this thesis are as follows:

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 multiple queries. In addition, the sub-expression granularity of the execution plan and the logical plan is different. We mitigate these issues by adding an additional parsing phase after the database created an execution plan.
2. **Execution Plans Enrichment** – Execution plans include statistics for our queries. unfortunately, some useful statistics are missing, and existing statistics may have an unintuitive representation and granularity. We mitigate these issues by:

* Inferring and adding sub-expressions from existing ones (like whether a sub-expression is redundant or not).
* Adding sub-expressions’ statistics with a more intuitive granularity (not cumulative) and a more intuitive representation (like percentage).

1. **Execution Plans Visualization** - A new representation for queries as a Sankey-diagram that allows us to understand the nature of a query, multiple queries, or even the optimizer itself. It can be used to find the cardinality issues, bottlenecks, and optimizer problems.

##### 1.3 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 identify flaws in queries using Sankey-diagrams.
* Section 4 provides 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 provides insights into common flaws, like as performance bottlenecks and cardinality issues. In this chapter, the relevant background and related work are provided.   
  
The rest of this chapter is structured as follows:

* Section 2.1 provides an overview of the common problems in SQL queries.
* Section 2.2 provides an overview of the execution plan.
* Section 2.3 cover related work on static analysis of execution plans.
* Section 2.4 cover related work on the debugging approach to identify flaws in SQL queries.
* Section 2.5 cover related work on SQL queries visualization approach to identify errors in SQL queries.
* Section 2.6 provides a comparison between the different approaches to identify errors in SQL queries.
* Section 2.7 covers related work on multiple queries optimization.
* Section 2.8 provides an overview of the Sankey Diagram and its merits. Section 2.9 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 area of research[1] 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 flaw can cause the following:

1. Wrong decision making.
2. Missed business opportunities.
3. Increased DBMS cost.
4. Bad experience of DBMS’s internal users, and waste of their time.
5. Bad customers’ experience due to slow applications.
6. Many more.

The first group of flaws corresponded to a unexpected query’ results. This group consists 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.
8. Many more.

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 spill to disk.
* Stale statistics can affect the optimizer to pick sub-optimal execution plan.
* Operations that don’t change the result.
* Operations that help several queries’ performance, but harm others.
* Many more.

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

* Static analysis of the query characteristics.
* Adding debugging capabilities to see intermediate steps.
* Visualization of the query.
* Query builders.

##### Most of the approaches taken are using the execution plan behind the seen, in order 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 parsed and validated the query. If the query is valid then it translates it to a tree representation, called execution plan. The execution plan is a sequence of steps used to [access data](https://en.wikipedia.org/wiki/Data_access) in a [SQL](https://en.wikipedia.org/wiki/SQL) [relational database management system](https://en.wikipedia.org/wiki/Relational_database_management_system) and to provide the users the result for his/her query.

Due to SQL’s declarative nature, several plans can be derived from a single SQL statement [2], 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 improve the query’s cost tremendously. Unfortunately, finding the best plan is not trivial, in fact [3] 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 type 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 plan – 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 plan), like the time and the *Actual Rows.*

PostgreSQL provide the *EXPLAIN* command, which is very 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 plan. By default, *EXPLAIN*  return the logical execution plan by default, and by adding *ANALYZE* it will return the actual execution plan.
* The user can add additional statistics using *BUFFERS, COST* and more.

Since the execution plans is flexible and include useful information at the sub-expression level, most of the related work that are covered in section 2.3, section 2.4 and section 2.5 will utilize either the actual execution plan or the logical execution plan to identify and fix the flaws in our queries.

##### 2.3 Static Analysis

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

* The empty answer problem - This problem occurs when a user writes restricted query that eliminate all the results.
* 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.
* What data change affected my query- providing debugging capabilities for the data itself.

The papers that tackle the “empty answer problem” focus on generating a less restrictive version of the query, such that a none-empty result is returned to the user. For example, IQR [4] uses a probabilistic framework to rewrite the query in such a way that it will return more result. It generates potential candidates for relaxation, for example, if when we remove a *WHERE* clause we get none-empty result, and 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. The IQR steps that are used can be seen in Figure 1.

Graphical user interface, application

Description automatically generated

**Figure 1**  
Although, tools that solves the “empty answer problem” can be useful, they have 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 user encounter.
* Require query execution.

The papers that tackle the “why and why not problem” focus on understand the flow and causes of results to appears or be missing in the query result. For example, Ned Explain [5] solve 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 miss the expected entry.

Although, tools that solves the “why and why not problem” can be useful, they have 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 user encounter.
* These systems tend to be complex and hard to maintain.
* Require query execution.

The last technique that uses static analysis is “what data change affected my query”. Since data debugging occurs before query debugging, we can consider data debugging to be orthogonal to identifying flaws in the query itself. For example, QFix [6] use the query log to look at past queries and to identify the ones that contributed to the errors. In Figure 2, we can see how QFix uses the query log to understand that Q1 caused the unexpected behavior.

Graphical user interface, text, application

Description automatically generated

**Figure 2**

Some of 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.

##### 2.4 Debugging Approach

In this section, we will review the relevant literature of debugging DBMS queries using view. These tools [7][8] 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.   
  
For example, Habitat [7] uses the user’s marking the desired sub-expression with a breakpoint and materialize the corresponding sub-expression to its own view. For example, we can mark the inner query with a breakpoint (s1) as can be seen in Figure 3, the view that was generated from this breakpoint can be seen in Figure 4.

Text

Description automatically generated

**Figure 3**

Table

Description automatically generated with medium confidence

**Figure 4**  
  
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.

##### 2.5 Query Visualization Approach

As we saw in section 2.2, both type of execution plans includes valuable information. In this section, we review the relevant literature that use visualization to understand the 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 plan to improve performance.
* Visualize the optimizer itself.

The first technique focuses on visualizing the logical execution plan of the query which give an intuitive understanding of a query. One of the most prominent advantages of using the logical structure is that by avoiding the execution of a query, it makes very scalable.   
  
For example, QueryViz [9] create a succinct represent of a query logical plan similar to ERD. In Figure 5 we can see how the query in the left is represented as an ERD like representation on the right.  
Diagram

Description automatically generated

**Figure 5**

Although, Visualize the logical execution plan can be useful, they have the following cons:

* It can provide only a shallow understanding of the query characteristics.
* They focus on query understanding and not identifying flaws.
* According to [12], ERD like visualization doesn’t have big impact the mental workload for understanding SQL queries.

The second technique focuses on visualizing the actual execution planof the query. These techniques give an intuitive understanding of the query and by observing how the query is executed under the hood and provides users with a much more granular understanding of the DBMS.   
  
For example, Perfopticon [11] help to understand the queries' bottlenecks in distributed databases by visualizing the overall execution plan and how the data flows among servers.  
In Figure [6], we can two types of visualization one that focus on how the physical plan and operations on different fragments and another that describe query characteristics for specific fragment.

Graphical user interface, application

Description automatically generated**Figure 6**

Although, Visualize the actual execution plan can be useful, they have the following cons:

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

The last technique focuses on visualizing the behavior of modern optimizers. Database optimizers produce numerous execution plans for each query, and the ability compare between optimal execution plan and the one selected is priceless and visualizing it in an intuitive manner can be useful.   
  
For example, Picaso [12] takes as input a query, and an optimizer, and generates several visualizations that help understand characterize the behavior of the optimizer. We can understand compare optimizers compilation time, or the plans they generated. Figure [7] we each colored region represents a specific plan, and how it covers the selectivity space.

Chart

Description automatically generated

**Figure 7**

We can see that each type of query visualization has pros and cons and knowing what to visualize and how can be very useful.  
  
In the next section I am going to cover a comparison for the approaches for debugging SQL and identifying flaws in SQL queries [13].

##### 2.6 Approaches Comparison

In the previous sections (2.3, 2.4, and 2.5), we provided an overview of approaches that can help identify and fix flaws when using DBMS. In this section, we are going to cover the conclusions from a survey of tools for debugging database queries [13].

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 technique was low, as only 4 out of 20 knew they even exists.

The participants stressed that DBMS errors and execution plan 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 is trial and error. They wrote their query, and then manually review the raw results from the DBMS, a wasteful and error prone approach. In order to find flaws in complex and nested queries, they tend to simplify the query and to divide it to several components, which make things even more wasteful.

The participants found static analysis and the debugger were less intuitive and effective, 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 identifying flaws in their queries. Most suggestions involve displaying intermediate results of the queries, which highlight the trace of where certain tuple came from. It important to know that some visualization techniques, such as generating ER diagrams was not found helpful.
* **Unexpected result indicator** – all the participants mentioned that summary statistic can benefit them when they query result was unexpected. Even though statistics seems to be simple, none of the tools they tried 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 flow in SQL query, 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 query optimizations.

##### 2.7 Multi Query Optimization

As organizations become more data-driven, having complex queries is the new standard. In order to speed up a complex query, some optimizations to database or the query itself may be needed . But optimization that may benefit one query may have negative affect 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 MQO problem. In those techniques, the optimizer can evaluate several queries together, such that the queries can share state and save repeated computation. Volcano [14], introduce several cost-based heuristic algorithms that decide what sub-expressions should be materialized and shared using DAGs. Rule based framework [15], that incorporates a set of new abstractions, that allow to integrate new and existing MQO techniques through the use of transformation rules.  
  
Unfortunately, MQO problem is much broader then how to share state effectively across multiple queries. When a user optimizes his query, it may have negative effect on others, such behavior will cannot be handled in the optimizer. For example, an index will make a scan faster but can have negative effect on updating a relation.   
  
For this reason, the ability to visualize multiple queries together is valuable. There is

work [16] that visualizing the result of multiple queries to together, to allow easy comparison. But technique like focus on data exploration and not on identifying bottlenecks like QueryFlow.

In section 2.5 we covered visualization techniques to identify problems in SQL queries. Unfortunately, these techniques often lack the ability to compare statistics of sub-expression characteristics, while keeping the sub-expressions hierarchy clear.

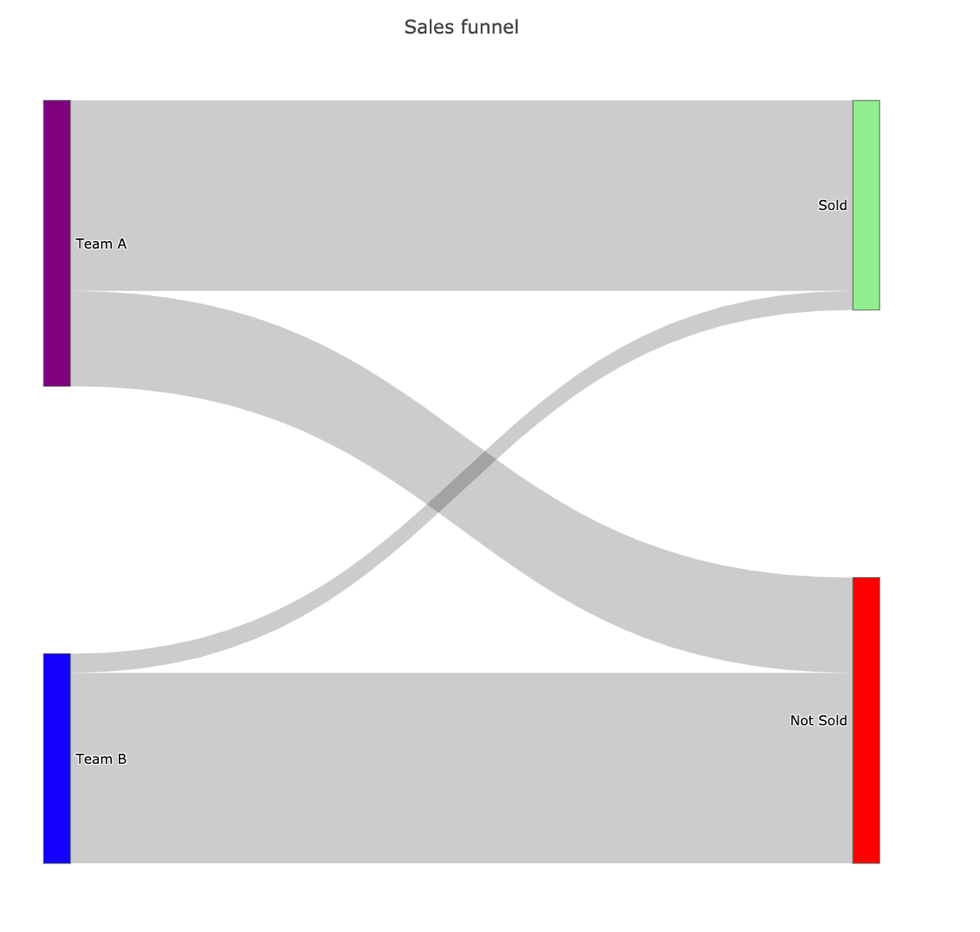
Sankey-diagram is a visualization technique that can mitigate it and I am going to cover it in section 2.8

##### 2.8 Sankey Diagrams

Sankey-diagram 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 visualized as a colored rectangle. And the links represent a measurable metric and 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 perform 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 team, we will have an edge to both deals that are sold and deals that are not sold, where the thickness represents the number of deals.

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

**Figure 8**

We can clearly 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, by simply hovering the edge. This can be seen in Figure 9.

****

**Figure 9**

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 edge 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. Energy ecosystems [17], that uses Sankey diagram to understand the utilization of fuel across different, devices, and end uses as can be seen in Figure 10.

Diagram

Description automatically generated   
 **Figure 10**  
  
Cancer research [18], that uses Sankey diagram to understand the number of patients for potential pathways and targets across different cancer types as can be seen in 11. 

**Diagram

Description automatically generated with low confidence**

**Figure 11**

In addition, since Sankey popularity is increasing, optimizing the Sankey-diagrams has become a research topic in its on right. In particular, techniques that enable a nicer layout, whether in terms of the number of crossing edges or their thickness. For example [17] uses integer linear programming to solve this. It models the problem by defining variables, constraints, and an objective function to solve NP-hard problems within reasonable time.

##### 2.9 Summary

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

The survey in section 2.6 clearly 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, which aid to understand how data flows.
* 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 plan, 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 behaviour on a sub-expressions granularity.  
  
QueryFlow is like a swiss-knife, it allows one to visualize both logical and actual execution plans for one or more queries. QueryFlow can help identifying both cardinality and performance problems in the same manner, by simply visualizing the relevant statistics.

In the next chapter, we will see QueryFlow design and understand how it works internally.

# 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 plan.   
  
This allow 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 return unexpected number of entries (Why and Why not problem).
  + Identifying duplication.
* Queries bottlenecks – finding the queries bottlenecks, for a single query or for 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 plan.   
  
QueryFlow process consists the following steps (Figure 12):

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.
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.
3. **QueryFlow visualization**- We start the visualization process by transforming the enriched parsed 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 emphase the query’s characteristics.

Diagram

Description automatically generated

**Figure 12**

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

##### 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 behaviour of the query. For example, we there is a statistic that corresponds to the number (and estimation) of records a sub-expression hold, another useful statistic is the execution time of the sub-expression (or estimation).   
  
In order 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 in order 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). In order to do so, we specify a hash function that indicates whether two sub-expressions from different queries are the same logically and give provide an indication for that later.

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 parsed execution plan on its own right, 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 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 subexpressions, 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 plan and planned execution plan only differ in the metrics themselves, apart from the redundant operation, all the enrichment types we described above are relevant. The reason the redundant operation is not enriched for a planned execution plan is that it’s an estimation that will bring more false positives and will lure the user from the real problems.

To enrich the statistics, I am using the algorithm in Figure 13 on the sub-expression graph:

Text

Description automatically generated

**Figure 13**

In the next section, we will how QueryFlow’s visualization phase take the enriched execution plan 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 subexpressions 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 subexpression. 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 regarding the operator are provided when hovering an edge.

In order 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).   
  
In cases where we want to visualize multiple metrics, we will represent them as two edges between the same nodes with a 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* and *label*  and the *value* and will differ in *variable* are different.

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* operation will be blue. By default, all the nodes are colored in black as it can be quite overwhelming due to number of different operations.

QueryFlow’s edges represent measurable statistics like cardinality or duration. To emphasis parts of the query that might need some special attention we will add coloring heuristics to the edges and nodes of the Sankey-diagram.   
  
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 exact 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 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.
* 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

The last set of heuristics is on the nodes and it help to distinguish between different types of relational operations.

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.

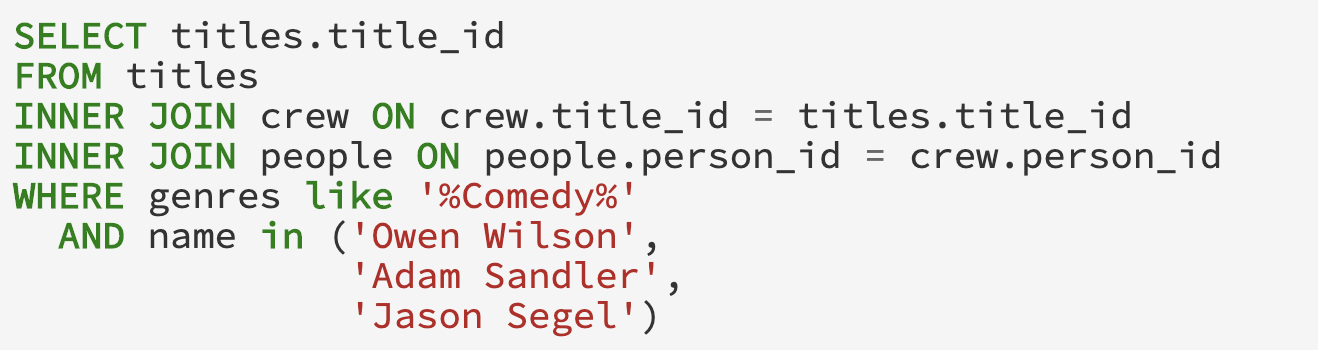
##### 3.5 QueryFlow Detailed Example

Through chapter 3 and chapter 5, 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 14.  
Diagram

Description automatically generated

**Figure 14**

We introduce a simple example, that is rich enough to illustrate the gist of the QueryFlow and how it actually 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 15.



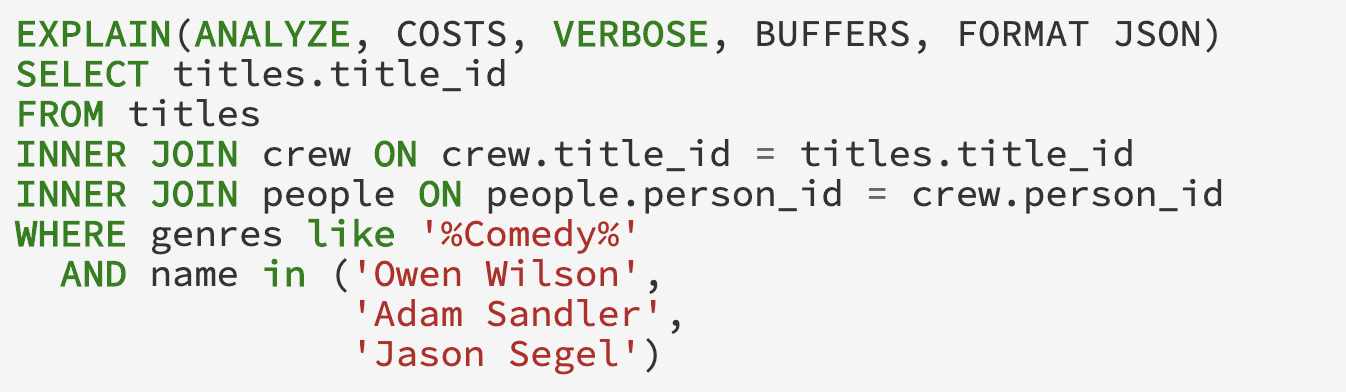
**Figure 15**

After we are given the query in Figure 15, 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 plan (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 plan.   
  
In both cases we will add to *EXPLAIN* clause the following attributes “*COSTS, VERBOSE, BUFFERS, FORMAT JSON”* in order to get addtional 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 16.

Text

Description automatically generated

**Figure 16**  
If we add the ANALYZE keyword, we will get the actual execution plan and we will get the query in Figure 17.



**Figure 17**

When executing the modified query in Figure 16 and Figure 17 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 plan can be seen in Figure 18. A picture containing text

Description automatically generated  
 **Figure 18**   
  
Just for the sake of comparison, the logical execution plan *JSON* can be seen in Figure 19 and it will have the same structure as the actual execution plan (Figure 18) apart from some missing statistics like the *Actual Rows*.   
  
Timeline

Description automatically generated with low confidence  
 **Figure 19**   
  
We can see in both Figure 18 and Figure 19 that the logical and actual execution plan 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 plan. There are a lot of different set of 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 actually is. For example, when we use *Seq Scan* we need to know which relation, and for that, we got *Relation Name.*

QueryFlow work in the same manner for both the planned execution plan and the actual execution time. From this step forward, so I will demonstrate the latter.

A more intuitive way to think about both the logical execution plan and the actual execution plan is as a tree structure. You can see the equivalent to Figure 18 in a tree representation (with parts of the information) in Figure 20.

**Diagram

Description automatically generated**

**Figure 20**

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.   
  
We start parsing our example in Figure 20, and in order to emphasis 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 and crew.
2. The *Hash-Join* operation has two ancestors, the *Seq Scan* of titles and the *Hash* operation. 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*.
3. The *Hash* operation has one ancestor, the *Hash-Join* between people and crew. Since we are looking for cardinality issues, and the *HASH* operator does not affect it, we can skip the hash and continue with the *Hash-Join* parsing.
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*.

Again, I will show the actual execution plan after parsing as the tree, and it can be seen in Figure 21.

Diagram

Description automatically generated

**Figure 21**

As you can see, it’s easier to understand the cardinality changes in every step of the query.

* We have separated *Scan* sub-expressions into two, one for the *Scan* and one for the filter.
* We can drop sub-expressions that don’t change the cardinality like the *Hash* sub-expression in the parsing step*.*

Now we got to the enrichment phase, as we still lack some can relevant information. We will enrich our execution plan in Figure 18 using the algorithm in Figure 13. The algorithm iterates the ancestor’s hierarchy as a BFS, and the iterations go as follows:

1. We run on the row which represents the *People* sub-expression.
   1. **Missing statistics in sub-expression granularity-** since this operation has no ancestors, no additional work 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 row which represents the *People\** subexpression.
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 where filtered.
6. **Human-readable representation**- *People\*.*
7. We run on the row which represents the *Crew* subexpression.
   1. **Missing statistics in sub-expression granularity-** since this operation has no ancestors, no additional work 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 row 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 row which represents the *Title* subexpression.
    1. **Missing statistics in sub-expression granularity-** since this operation has no ancestors, no additional work 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 row which represents the *Title\** subexpression.
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 where filtered.
18. **Human-readable representation**- *Title\*.*
19. We run on the row which represents the *People\* ⋈ Crew ⋈ Title\* subexpression*.
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\*.*

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**

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 row and the *target* is an identifier of one ancestor of the current row.
* **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 with medium confidence

**Figure 23**

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

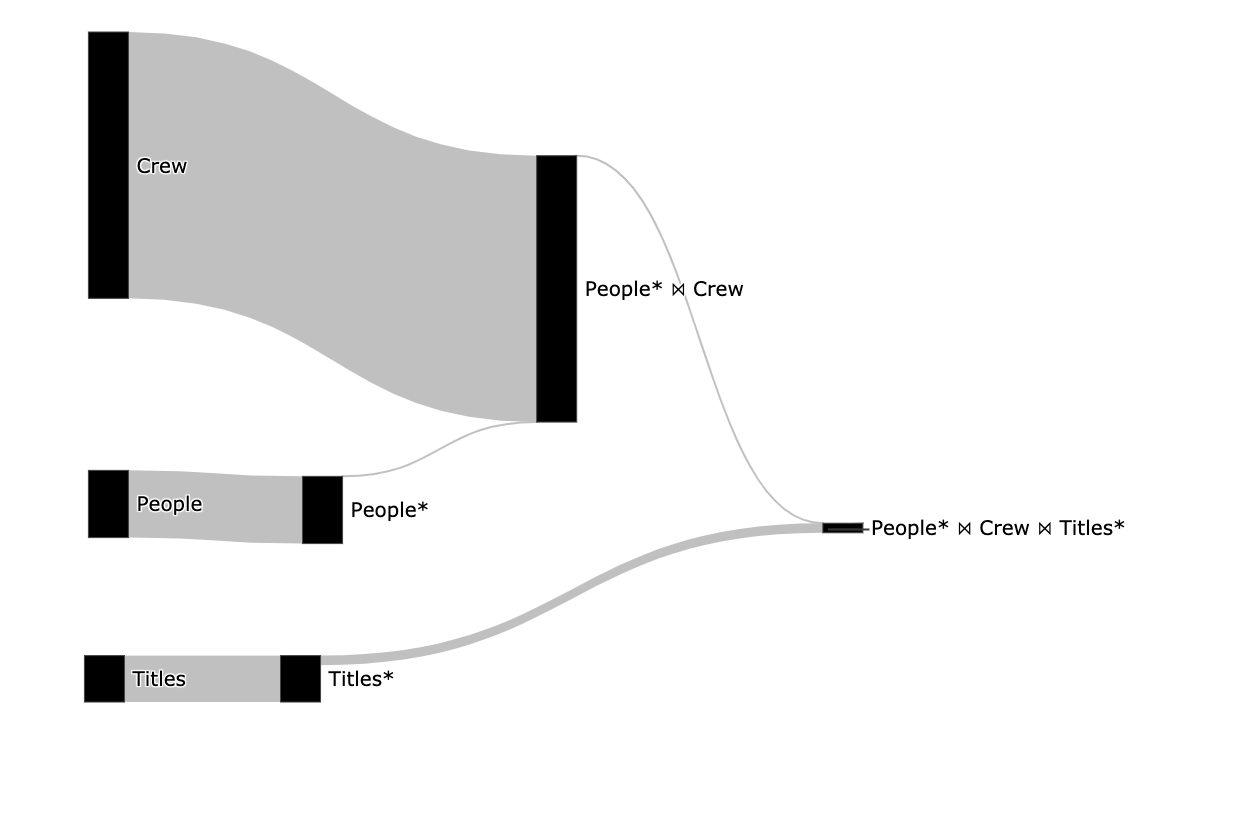
1. The node representing *People* is a table scan with 3,446,261 rows and has an edge to *People\**.
2. The node representing *People\** has 3 rows and has an edge to *People\* ⋈ Crew*.
3. The node representing *Crew* has 13,651,901 rows and has an edge to *People\* ⋈ Crew*.
4. The node representing *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* 186 rows and it is a terminal node.

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 a multiple query.
* We are investigating a multiple statistic.
* 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 their will be proportional to the cardinality (*actual\_rows)*. Additional details regarding each sub-expression are can be seen when hovering over an edge.



**Figure 24**

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 15 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.

In the next chapter, we will see the use cases QueryFlow can 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 plan. For this reason, it allows to identify a lot of different flaws, by simply visualizing the relevant statistics.

The rest of this chapter is structured as follows:

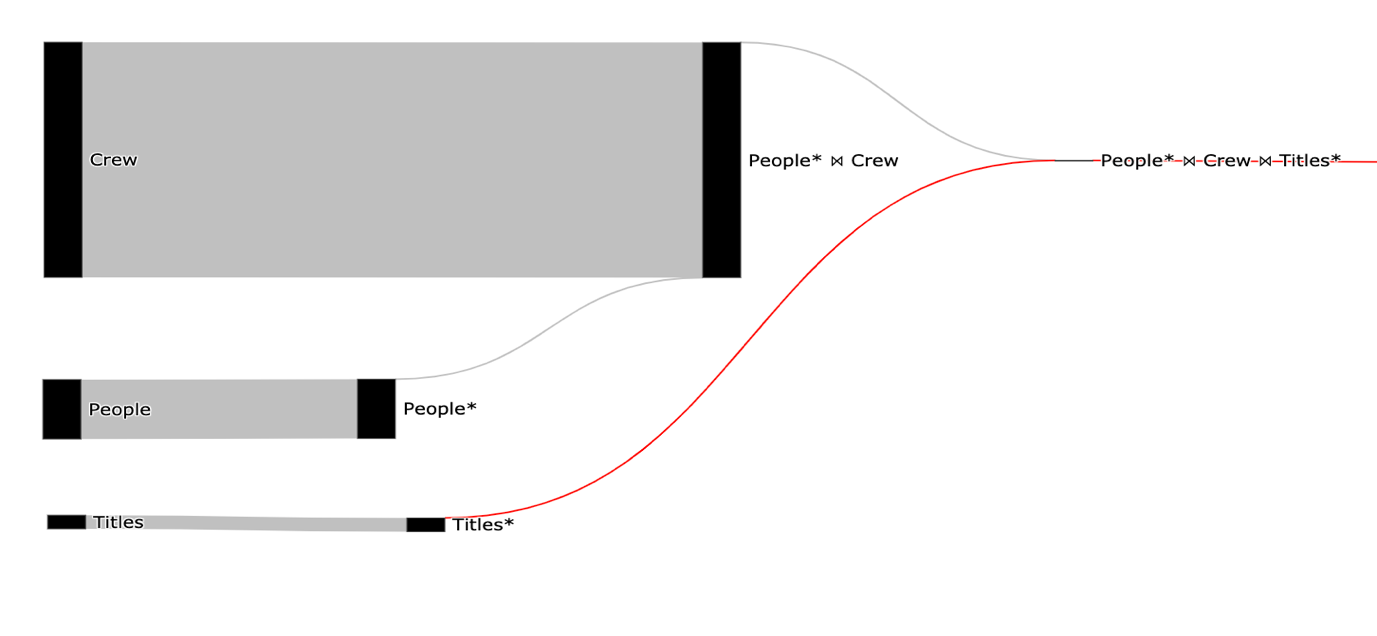
* Section 4.1 provides an example of identifying missing records using QueryFlow.
* Section 4.2 provides an example of identifying duplicated entries using QueryFlow.
* Section 4.3 Identifying performance bottlenecks in a single query using QueryFlow.
* Section 4.4 provides an example of identifying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

4.1 Identifying missing recordsProblems related to missing records are really common and finding them tends to be really 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 empty result. This problem can be caused due to either *WHERE, JOIN, UNION,* or *HAVING* clauses.

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 25.

****

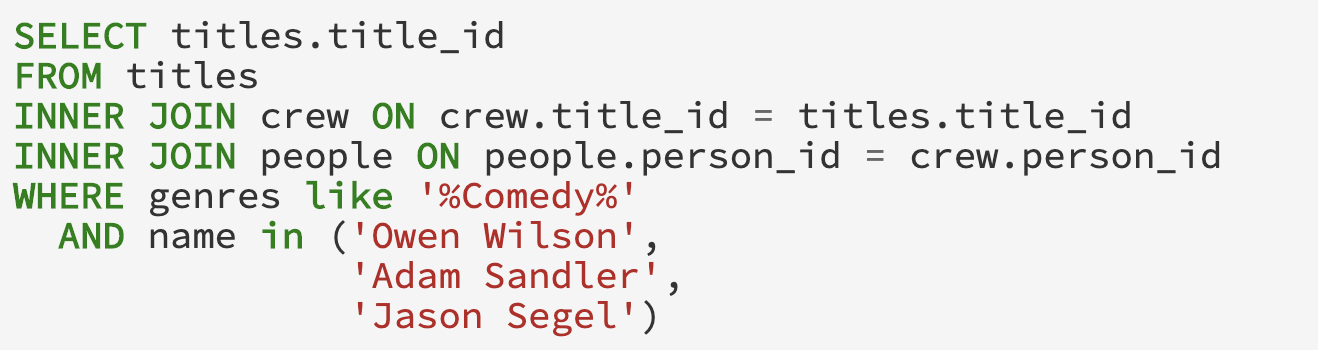
­­  **Figure 25**  
We are going to use QueryFlow to identify the empty results (will have red edge) and from which sub-expression it was originated. The corresponding Sankey in Figure 26represents the cardinality of our query’s sub-expressions.

****

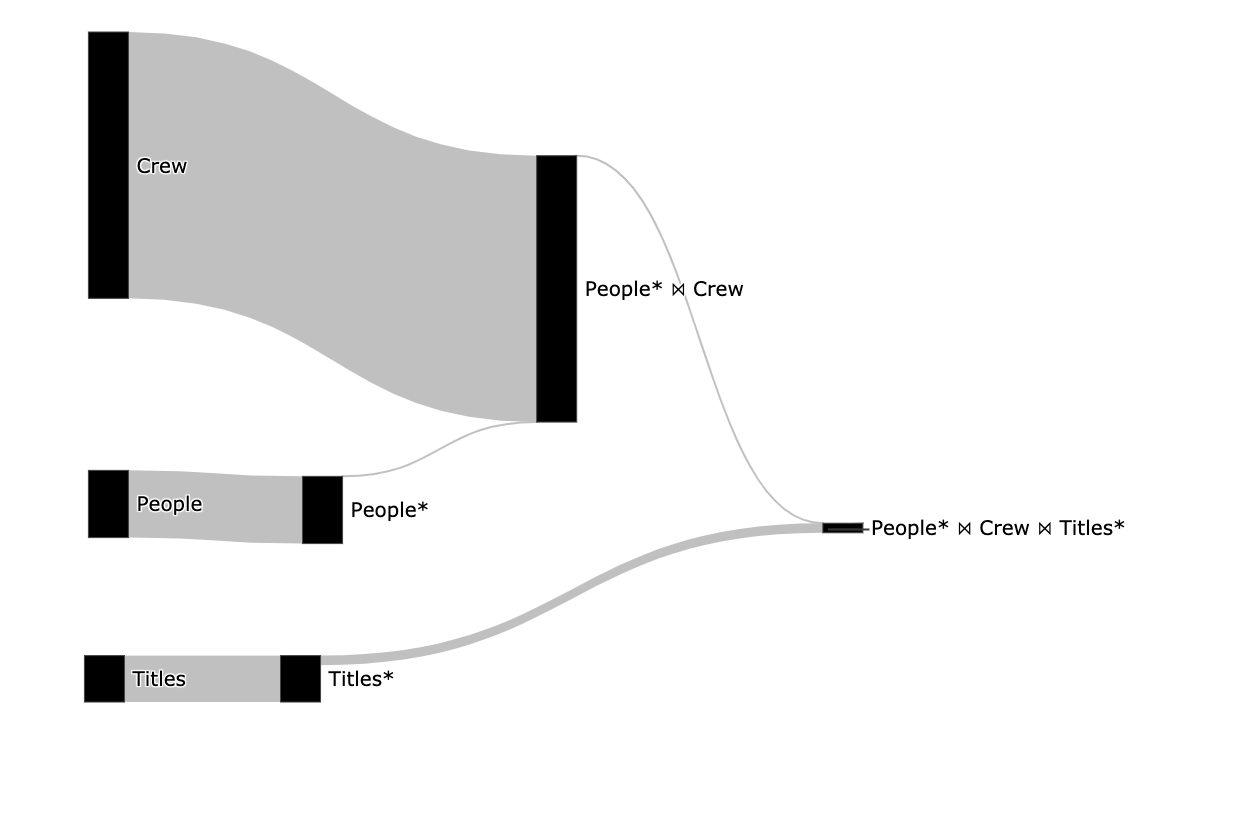
­­ **Figure 26**

Using the Sankey visualization, we can clearly see that we got an empty result in the *People\* ⋈ Crew ⋈ Title\** relation*,* and we can see the origin of the empty result 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 27.



­­ **Figure 27**  
To validate 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 28.

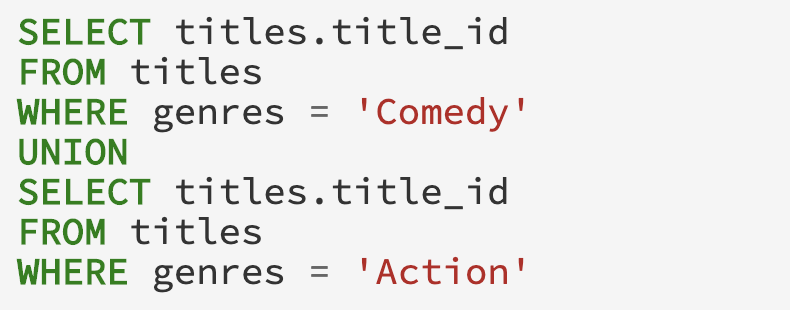


­­ **Figure 28**

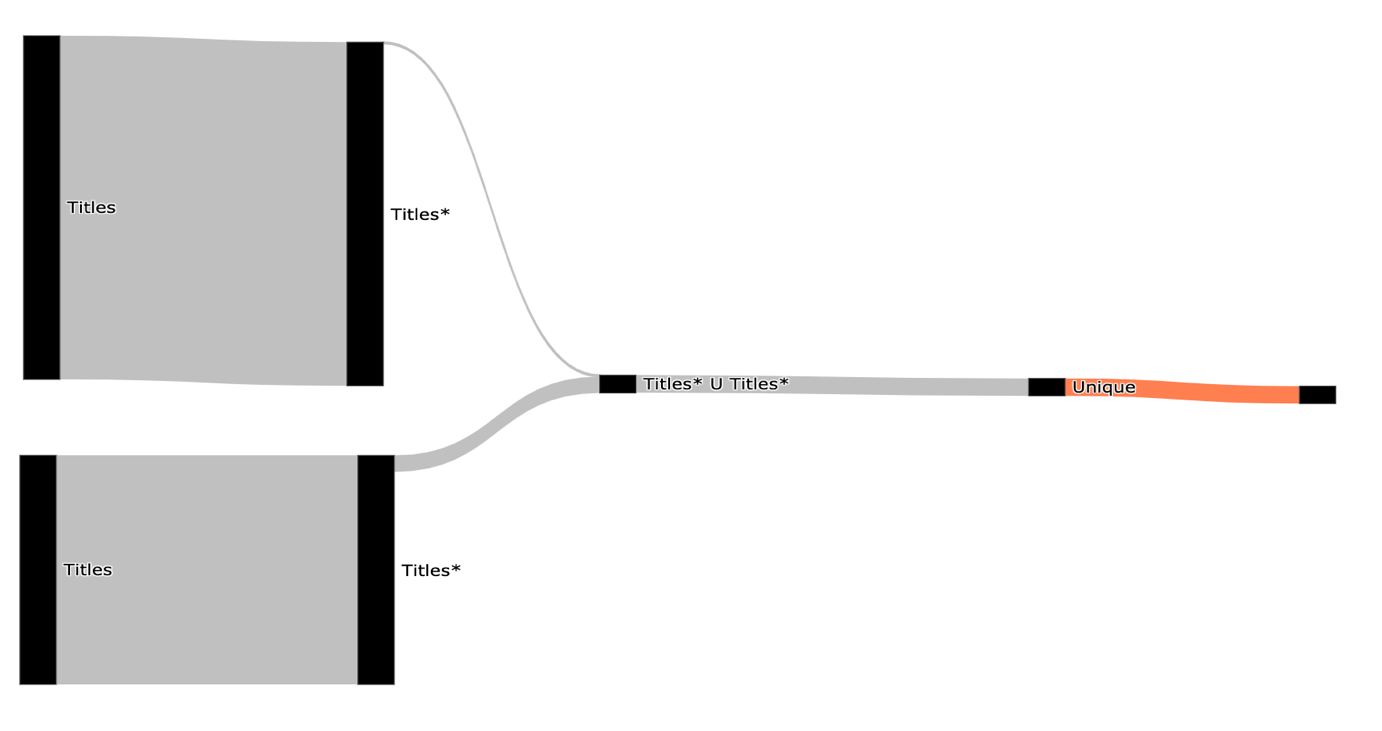
From this simple visualization in Figure 28 we can infer the following:

* 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 performance stand of point can enjoy optimization techniques like indexing and partitioning.

##### 4.2 Identifying Ineffective Operations

Problems related to infective operation are really common and finding them tends to be nearly impossible with the tooling today. This kind of flaws relate 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.   
  
Using QueryFlow we can visualize the cardinality (*actual\_rows)* of the query’ sub-expressions and find find ineffective operations. Ineffective operations are operations that should change the granularity of its direct ancestor but doesn’t, for example, a *UNIQUE* clause that filters nothing. This problem can be caused due to either *WHERE, JOIN, UNION,* or *HAVING, UNIQUE* clauses.   
  
In order to show an example of ineffective operations, we introduce a following question, “list all comedy movies and all action movies”, this question is equivalent to the SQL query in Figure 29.  
  


­­  **Figure 29**We are going to use QueryFlow to identify the redundant sub-expression (redundant operations will have orange edge). The corresponding Sankey that represent the cardinality of our example can be seen in Figure 30.

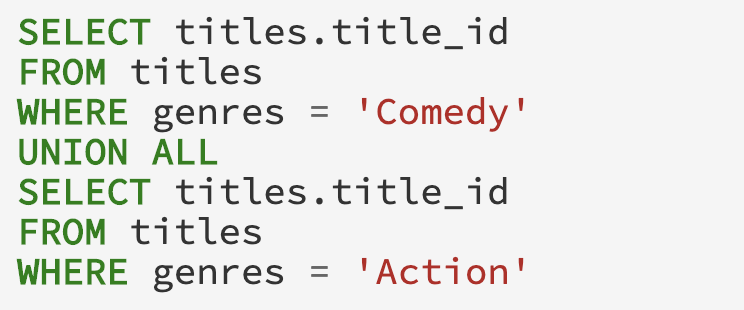


­­ **Figure 30**

Using the Sankey visualization, we can clearly see that the *Unique* sub-expression is redundant (it filters nothing) as it 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.

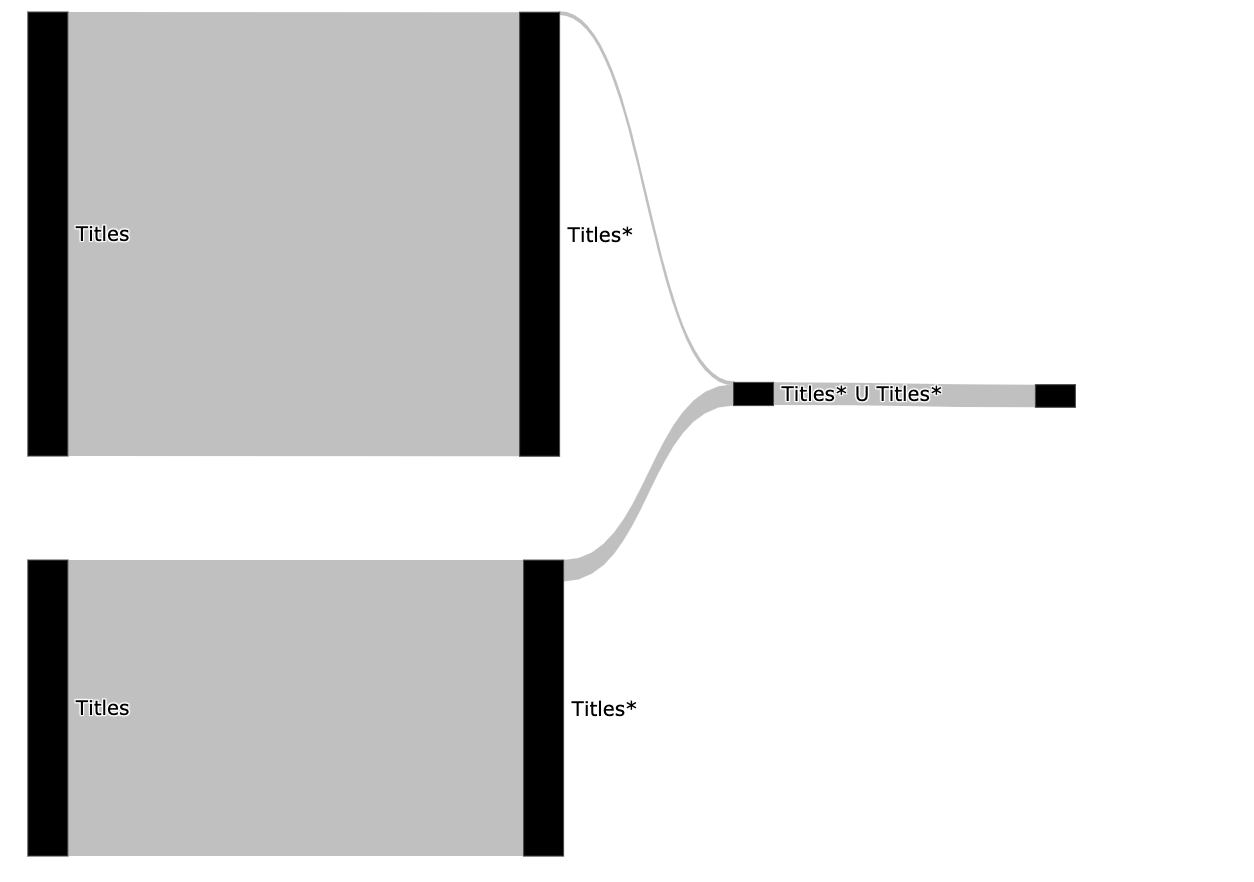
In PostgreSQL the difference between *UNION ALL* and *UNION* is that the first only append two relations and the second remove duplicates entries after the append. Since both relations are disjoint, there is no need for removing duplications after the append. This will not affect the query correctness but will improve the query performance as *Unique* is 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 31.



**Figure 31**

To validate 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 32.



­­ **Figure 32**

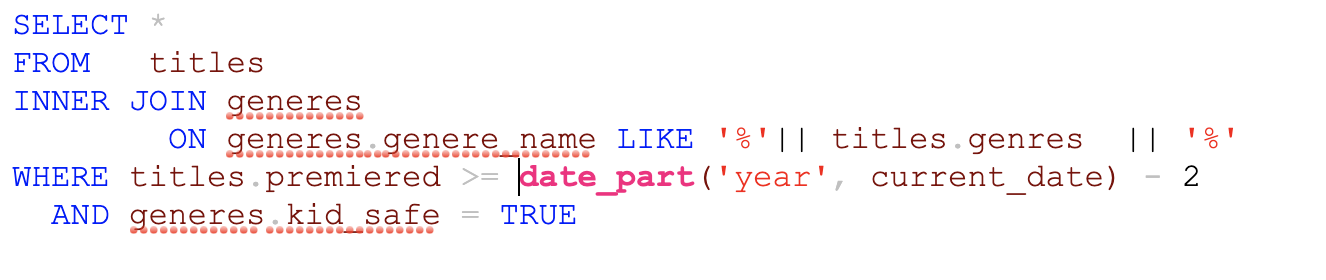
From this simple visualization in Figure 32 we can infer the following:

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

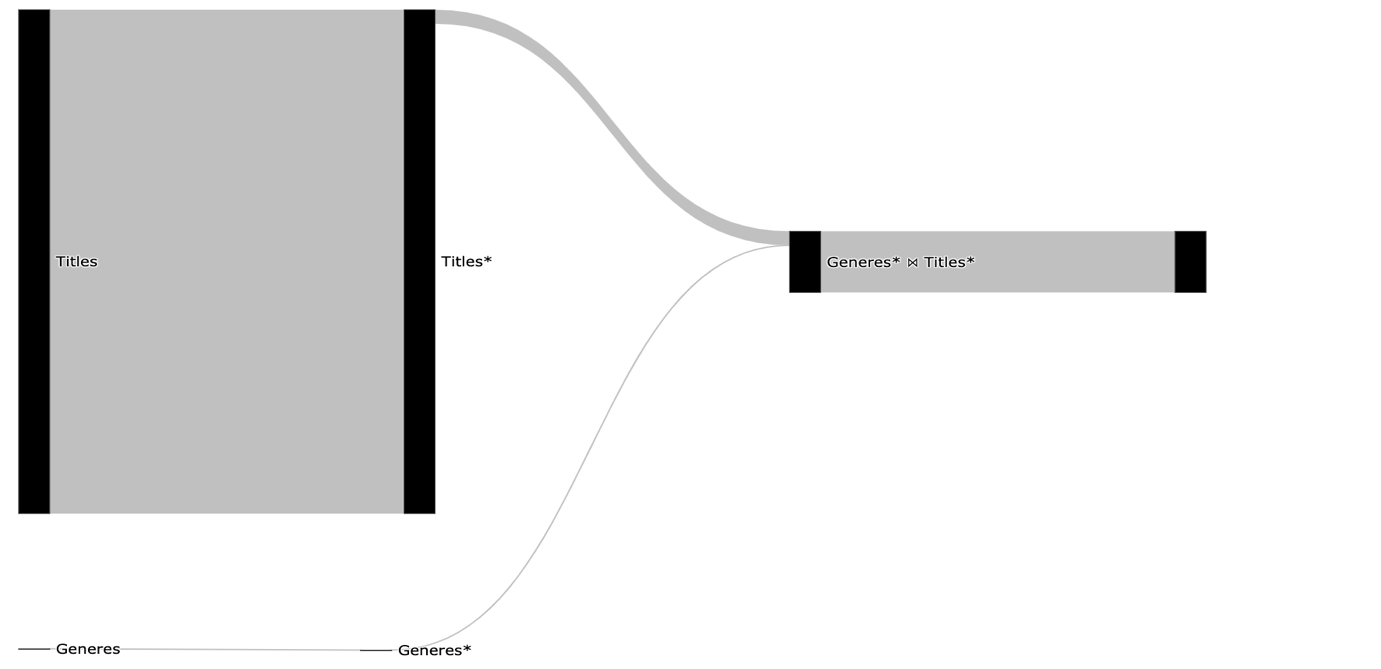
##### 4.3 Identifying Duplications

Problems related to duplicates entries are really 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.

In order to show an example of identifying duplications, we introduce a 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 33 ( the | | operation is a string concatenation in PostgreSQL).



­­  **Figure 33**We are going to use QueryFlow to identify duplications in our query. The corresponding Sankey that represent the cardinality of our example can be seen in Figure 34.



­­ **Figure 34**

Using the Sankey visualization, we can clearly see that *Join* subexpressions is exploding (its bigger than its direct ancestor and we expected one to one relationship). This allow us to understand the *JOIN* condition is wrong and cause duplicate records.  
   
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.*The query is the same as Figure 24 but we 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 35.

A picture containing text

Description automatically generated

**Figure 35**To validate that our query is actually 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 36.

Diagram

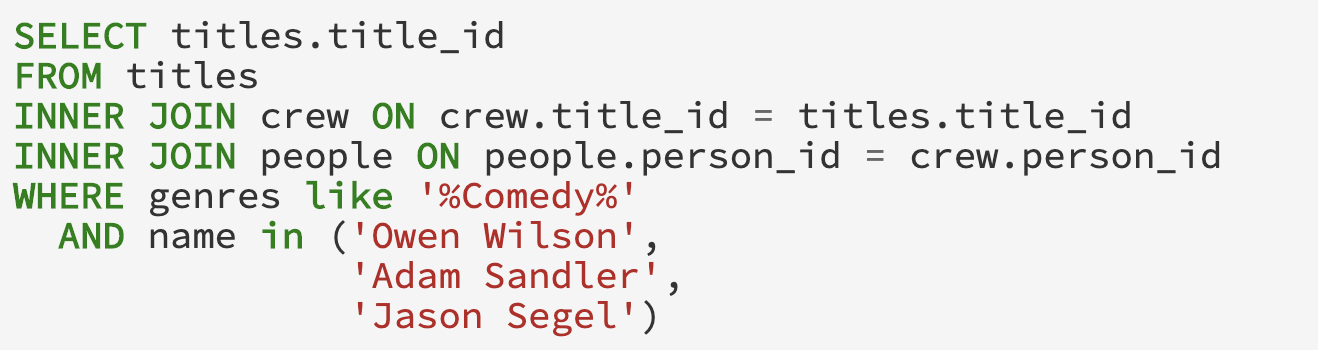
Description automatically generated

**Figure 36**

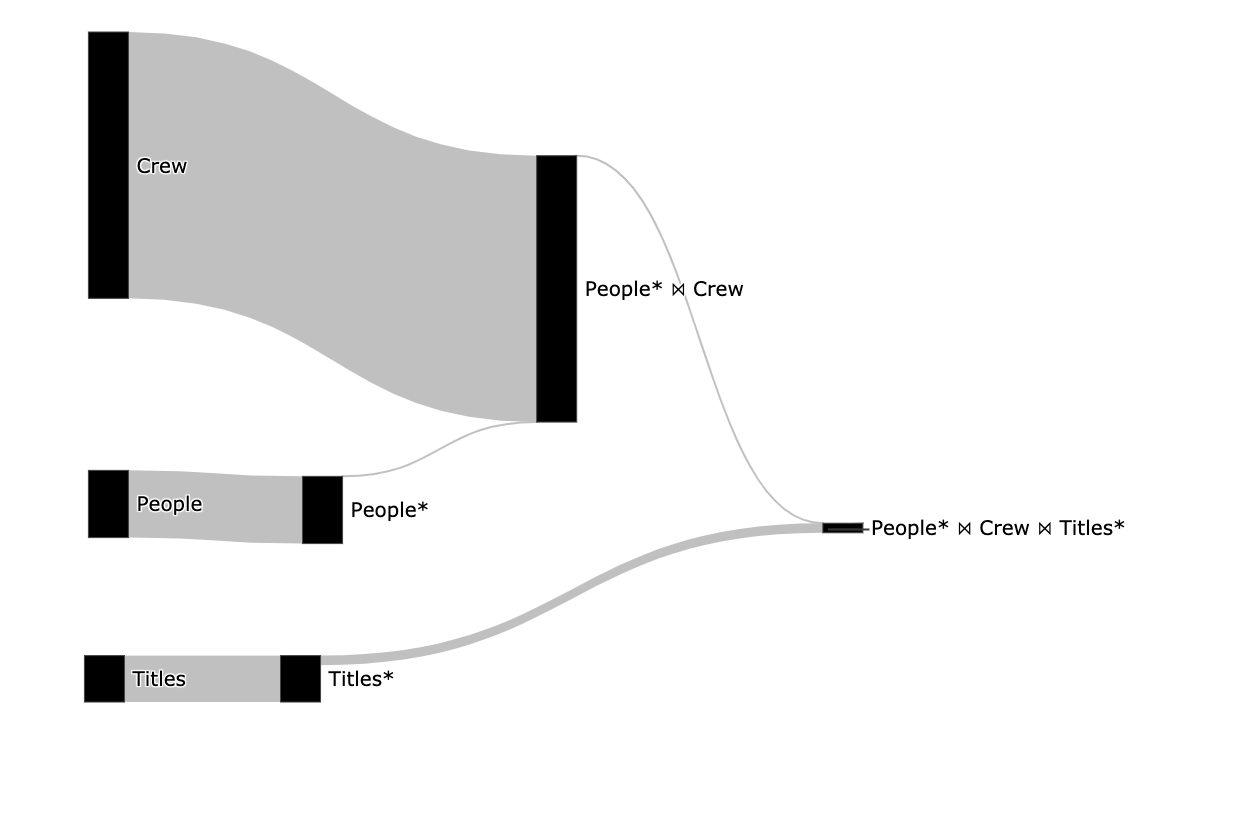
We can immediately see (Figure 36) the join still introduces the duplications but after the *Sub query* clause, we are removing those duplications in the *t\** sub-expression.

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

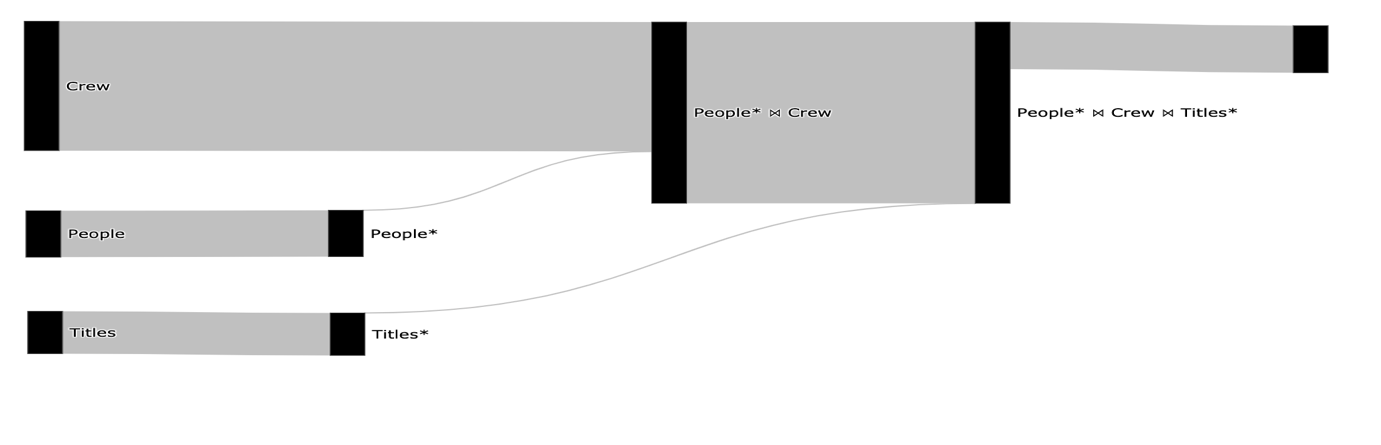
Slow queries are really common and can be caused for various reasons. This makes optimizing queries extremely hard. Using QueryFlow we can visualize the query’ sub-expressions characteristics.   
  
Since the execution time of an expression can be caused from several reasons, we might need to visualize the cardinality (*actual\_rows*) ,duration (*actual\_duration*) and other statistics like whether operation was spilled to disk.

We will use the same example as in chapter 3, the SQL can be seen in in Figure 37.  


­­ **Figure 37**We are going to use QueryFlow to identify the bottlenecks of our query.The corresponding Sankey that represent the cardinality can be seen in Figure 38.



­­ **Figure 38**  
We can see from Figure 38 that only a few rows are retrieved from the *People\* ⋈ Crew* due to the filter on the *People* relation, which indicate it might be a good candidate for optimization.   
In order to get more information, we are going to create another Sankey that represent the duration of our example, and it can be seen in Figure 39.

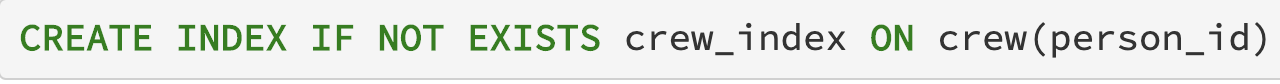


­­­­ **Figure 39**

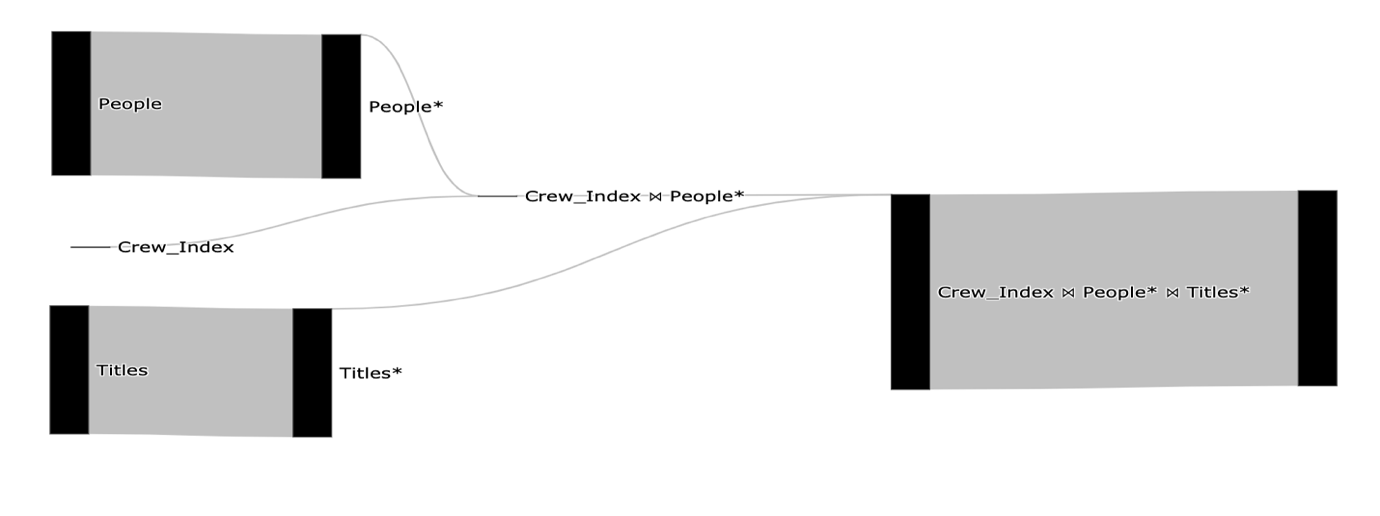
The total query duration is 7 seconds, we can see in Figure 39 that the longest operation (3.6 seconds) is the People\* ⋈ Crew.

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 40.



­­ **Figure 40**  
To validate 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 41.

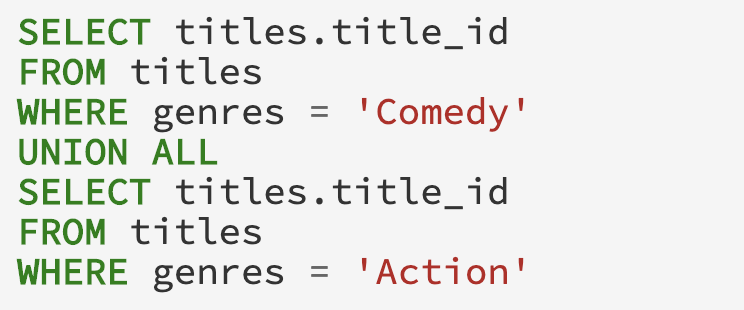
****

­­ **Figure 41**

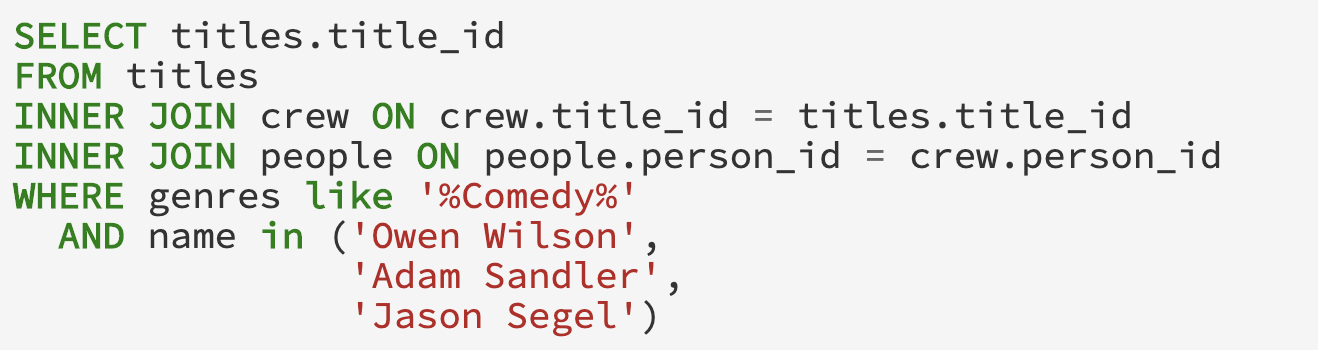
The total query duration is 2 seconds, we can see in Figure 41 that it improved both the *People\* ⋈ Crew* and the scan on the *Crew* relation.

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

4.5 Identifying Performance Bottlenecks in Multiple Queries   
  
Slow queries are really 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 an expression can be caused from several reasons, we might need to visualize the cardinality (actual\_rows) ,duration (actual\_duration) and other statistics like whether operation was spilled to disk.

In order to illustrate the gist of the MQO problem, we use a two simple queries. The first query is the same example as in 4.2. The query can be seen in Figure 42. 

­­  **Figure 42**  
The second query we will use is the same example as in chapter 3. The query can be seen in Figure 43.



­­  **Figure 43**

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 has filter on *comedy,* they will share both the *Title* node and the *Title* node that represent the *comedy* filter. The corresponding Sankey that represents the cardinality of our example can be seen in Figure 44.

Diagram

Description automatically generated

­­ **Figure 44**We can see from Figure 44 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 indicate these might be a good candidate for optimization.   
In order to get more information, we are going to create another Sankey that represent the duration of our example, and it can be seen in Figure 45.

Timeline

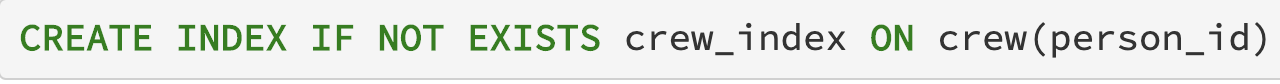
Description automatically generated­­

**Figure 45**

The total query duration of both queries is 7.5 seconds, we can see in Figure 45 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 46.



­­ **Figure 46**

To validate 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 47.

**Diagram

Description automatically generated with medium confidence**

­­ **Figure 47**

The total query duration is 4.5 seconds, we can see in Figure 47 that it improved both the *People\* ⋈ Crew* and the scan on the *Crew* relation.

Performance optimization is an iterative process. In order 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 48.   
­­ **Figure 48**  
To validate 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 49.

A picture containing diagram

Description automatically generated

­­ **Figure 49**

The total query duration is 3.5 seconds, we can see in Figure 49 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 its 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.

##### 4.6 Identify flaws in the optimizer itself

Problems related to the optimizer work are really hard to detect for regular users. Using QueryFlow we can visualize and compare the optimizer estimations to the actual statistics after executions.   
  
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 50.

Text

Description automatically generated

­­  **Figure 50**  
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 51.

Diagram

Description automatically generated

­­ **Figure 51**

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 51 that for 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 the PostgreSQL’s mechanism for deleting and updating records. When update or a delete occur it does not create an extra space in the system. PostgreSQL rather flag these tuples as *"dead tuple"* and in order to remove those, one need 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 52 .   
  
Graphical user interface, text, application, chat or text message

Description automatically generated

**Figure 52**To validate 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 53.

**Diagram

Description automatically generated**

­­ **Figure 53**

We can immediately see in Figure 53 that the *Crew* scan is no longer skewed as used to be, as the darker and lighter edges of the *Crew* sub-expression is pretty proportional.

##### 4.7 When QueryFlow won’t help

In 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 visualize query sub-expressions’ according to a measurable metric. Thus, if we have very skewed values, some of insights that can be inferred will be less visible.   
  
For example, in Figure 54 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 54**  
In order 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 55.  
A picture containing diagram

Description automatically generated

­­ **Figure 55**  
  
In addition, when we have a very complex Sankey-diagram one can be overwhelmed with information. In Figure 56 I show and example 100 teams with 25 sales phases and we can see that it’s really hard to make sense which ones are bigger.

A screenshot of a computer

Description automatically generated with low confidence

­­ **Figure 56**

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 relays 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 on PostgreSQL and I will use Dexter as a baseline.

The rest of this chapter is structured as follows:

* Section 5.1 provides an overview of TPC-H
* Section 5.2 provide an overview of evaluation setup.
* Section 5.2 evaluate optimizations for scale factor 1.
* Section 5.3 evaluate optimizations for scale factor 10.
* Section 5.4 provide 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 57.  
   
Diagram

Description automatically generated  
­­. **Figure 57**

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 to 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 in order to make a fair database engines comparison. This is understandable, as allowing tricks like materialised 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. In order to load TPC-H data, we are going to use dbgen to generate CSV files representing our tables and we then we 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 in order 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 [18, 19]. 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 Figure 58.   
 Chart, histogram

Description automatically generated **Figure 58**

##### 5.2 Evaluate optimizations for scale factor 1 As we described TPC-H consists of a suite of business-oriented ad-hoc querie 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 in order 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 59**

Figure 59 can be overwhelming, as we have a lot of nodes and edges. We can clearly see that there few long queries, that takes most of the execution time. Thus, we will benefit from checking only 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 60.

Timeline

Description automatically generated

**Figure 60**

Now it much easier to understand the heaviest parts of the queries. The first thing we would like to check is whether its 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 show when we over we can see there were no spill to disk and our 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

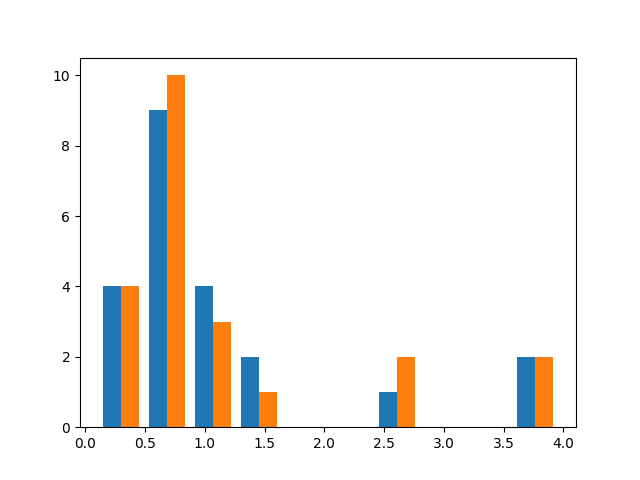
As a reminder before the optimization TPC-H took 22 seconds and the histogram can be seen in Figure 58, after the optimization we got to 14 seconds and you can see the histogram in Figure 61

Chart, histogram

Description automatically generated

**Figure 61**

For a comparison between the baseline and the optimization can be seen in Figure 62



**Figure 62**

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

##### ~~We will use QueryFlow to identify the bottlenecks across TPC-H’s queries in order 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 in a compact manner.

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 spills 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 filtering of unreverent subexpressions or queries.

# References

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

[2] 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

[3] 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>

[4] 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).   
[5] 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).

[6] 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.

[7] 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)

[8] [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)

[9] [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)

[10] [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.

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

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

[13] 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

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

[15] 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

[16] 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.

[17] 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.

[18] Bhattacharyya R, Ha MJ, Liu Q, Akbani R, Liang H, Baladandayuthapani V. Personalized   
 Network Modeling of the Pan-Cancer Patient and Cell Line Interactome. JCO Clin Cancer   
 Inform. 2020 May;4:399-411. doi: 10.1200/CCI.19.00140. PMID: 32374631; PMCID:   
 PMC7265783.

[17] 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

[18] 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.

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