**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

Table of Contents

[Abstract 3](#_Toc62286986)

[Chapter 1: Introduction 4](#_Toc62286987)

[Chapter 2: Background and Related work 5](#_Toc62286988)

[Chapter 3: QueryFlow Design 11](#_Toc62286989)

[Chapter 4: QueryFlow Use-cases 28](#_Toc62286990)

[Chapter 5: Evaluation 45](#_Toc62286991)

[Conclusions and Future Directions 49](#_Toc62286992)

[References 49](#_Toc62286993)

# 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 a better way to identify flaws in SQL queries. We want that our solution can be easily integrated with the existing

real-world systems and therefore assume that we cannot modify the databases themselves. Our

solution is implemented solely on the database parser output.

We have implemented our solution called QueryFlow, a query visualization tool that provides insights into common problems, such as performance bottlenecks and cardinality issues. The experimental results show that our solution allows us to pinpoint and fix the flaws in our queries.

# Chapter 1: Introduction

SQL is a powerful declarative query language, designed for managing and manipulating data, and for decades SQL has been the main 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.  
  
Finding 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. 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 subexpression to its corresponding part in the execution plan is difficult for most users.

To help users identifying their problems and to provide a performant solution, techniques to visualize the execution plans were invented. These techniques give the users a much more intuitive understanding of our 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 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. **Enrichment Multiple Queries Execution Plan Consolidation** - 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.

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 data

finding flaws in queries using Sankey-diagrams. Section 4 provides different flaws that can be detected using QueryFlow. Section 5 introduces 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, such as performance bottlenecks and cardinality issues. In this chapter, the relevant background and related work are provided.   
  
Section 2.1 provides an overview of the common problems and their effects. 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 errors 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 errors in user’s queries.   
  
These flaws can originate from problems in the data or in the query itself. In this section, I am going to cover the most common groups of flaws and how they affect DBMS’s users daily.

The first group of flaws corresponded to a unexpected query’ results. 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 manner. To make things worse most of these flaws are not identified and may result in wrong decision making. This group of flaws includes but not limited to the following examples:

* The data is not as expected.
* Query result with zero entries due to a wrong filter.
* Filters that remove no entries.
* Join between relations that return fewer entries than expected.
* Join between relations that return more entries than expected.
* Join between relations that return duplicate entries.
* …

The second family of flaws corresponded to queries’ performance, bad performance can affect DBMS’s users in many ways.

* The query execution affects the customer’s experience. In today’s world, customers expect everything as fast as possible, a slow request can lead to customers churning.
* The query execution affects the analyst’s experience. Slow queries will reduce analyst’s productivity tremendously.
* DBMS are expensive and using unoptimized queries will increase the hardware cost tremendously.
* …

This group of flaws includes but not limited to the following examples:

* Missing indices.
* Unused indices.
* Inefficient operations like scans, joins, aggregations, sort and distinct.
* 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 of the approaches to solve and identify these flaws using the execution plan. In order, to understand those better in section 2.2 I will provide an overview.

##### 

##### 2.2 Execution Plan Overview

A  query 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). Due to SQL’s declarative nature, a SQL query can be executed in several different ways, and each will produce the same results.

The planner/optimizer’s job is to create an optimal execution plan. Choosing the right plan to match the query structure and the properties of the data is critical for good performance, so the system includes a complex *planner* that tries to choose good plans.

We can use the *EXPLAIN* command to see what query plan the planner creates for any query, and support *TEXT*, *XML*, *JSON*, *YAML* as an output format. *EXPLAIN*  retrieve the expected execution plan by default, by adding the *ANALYZE*  keyword the DBMS’s the query will be executed and more statistics about the intermediate sub-expressions will be added.

The planner can create the output itself has one line for each node in the plan tree. Each of these nodes has 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.
* The estimates of the planner, like the cost and the *Plan Rows.*
* Extra statistics in case we added the *ANALYZE*  keyword.

Understanding the behavior and performance of individual plan nodes is critical to understanding the query behavior.  
  
For this reason, the approaches in section 2.3, 2.4, 2.5 utilize the execution plan to identify and fix the query.

##### 2.3 Static Analysis of the Execution Plan

In this section, we review the relevant literature in static analysis of the query execution plan. As we saw the section 2.2, by statically analyzing the execution plan we can understand the query behavior. There are several techniques that utilize it to identify and fix flaws in a SQL query each focus on a different use-case.

The first technique focuses on “empty answer problem”. This problem occurs when a user writes a very restrictive query that eliminate all the results. The techniques that tackle the “empty answer problem” focus on generating a less restrictive version of the query, such that results are returned to the user. For example, IQR [2] uses a generative probabilistic framework that generate a relaxation and rewrite a less restrictive query. It uses the probability of the user accepting a suggested relaxation, as well as other optimization objectives, such as, minimizing the number of user interactions or returning relevant results as can be seen in Figure 1.

Graphical user interface, application

Description automatically generated

**Figure 1**  
These techniques can be useful, but they have substantial problems that make them less appealing. The modified query might not reflect the user’s intention anymore. In addition, the “empty answer problem” error is only a fraction of the errors that user encounter.

The second technique focuses on “why and why not problem”. This technique can help understand why a specific appears in the result or why it doesn’t appears in the results. The techniques that tackle “why and why not problem” iterating over the sub-expressions and check whether a specific value exists or missing in that sub-expression. For example, Ned Explain [3] answer the “why not” problem by tracing the sub-expressions to identify points the query where key results are eliminated and filtered.

These techniques can be useful, but they have substantial problems that make them less appealing. The modified query might not reflect the user’s intention anymore. In addition, the “empty answer problem” error is only a fraction of the errors that user encounter. Lastly, these systems tend to be complex and hard to maintain.

##### 2.4 Debugging Approach

In this section, we review the relevant literature of debugging DBMS. The debugging approaches to finding and identifying flaws. There are several techniques that allow debugging.

The first technique focuses on providing debugging capabilities for the SQL queries. These tools [4][5] allow user to add breakpoints capabilities and retrieve the result of the sub-expression using views. These debugging tools give the users a much more granular understanding of the query flow. On the other hand, in order to identify where the flaw originated it require to consume much more resources than other strategies. In addition, these techniques seem to be very specialized and will now work on your production database. For example, Habitat [4] uses the user’s marking and collects the desired sub-expression as a view, which means that every breaking point creates a new view, which can drastically affect the performance of complicated query with several breakpoints as can be seen in Figure [2/3].

Text

Description automatically generated

**Figure 2**

Table

Description automatically generated with medium confidence

**Figure 3**  
The second technique focuses on providing debugging capabilities for the data itself. Since data debugging occurs before query debugging, we can consider data debugging to be orthogonal to query debugging. For example, QFix [6] use the query log to look at past queries and to identify the ones that contributed to the errors. we can use the query history to explain how errors occur in the database like, but it's will not provide a solution for issues in the actual query itself and thus quite limited as can be seen in Figure [4].

Graphical user interface, text, application

Description automatically generated

**Figure 4**

##### 2.5 Query Visualization Approach

As we saw in section 2.2, the query logical structure and execution structure can be very useful. There are several techniques that utilize them and visualizing it in an intuitive manner.  
  
The first technique focuses on visualizing the logical structure 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. On the other hand, it can provide only a shallow understanding of the query characteristics. For example, QueryViz [7] create a succinct represent of a query logical plan similar to ERD as can be seen in Figure [5].  
Diagram

Description automatically generated

**Figure 5**

The second technique focuses on visualizing the execution structure of the query. These techniques give an intuitive understanding of the query and by observing how the query is executed under the hood it provides users with a much more granular understanding of the DBMS. But since we actually executing the queries it might become resource-heavy for big data use-cases. For example, Perfopticon [8] help to understand the queries' bottlenecks in distributed databases by visualizing the overall execution plan and how the data flows among servers as can be seen in Figure [6].

Graphical user interface, application

Description automatically generated**Figure 6**

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 a intuitive manner can be useful. For example, Picaso [9] takes as input a ‘query, and an optimizer, and generates a suite of diagrams that characterize the behavior of the optimizer over this space as can be seen in Figure [7].  
Chart

Description automatically generated

**Figure 7**

There are various of research papers that check the effectiveness of visualization for SQL queries. Although, visualization doesn’t impact the mental workload for understanding SQL queries [10], it does have impact when trying to identify flaws in SQL queries [11].

##### 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 in DBMS. In this section, we are going to cover the conclusions from a survey of tools for debugging database queries [11].

The study conducted included 20 participants, including 6 undergraduate students, 5 graduate students and 10 industry professionals. Awareness of database debugging technique was low, only 4 out 200 knew they exists.

According to the survey most participants use trial-and-error. They are writing their query, and then manually review the raw results from the DBMS, which can wasteful and error prone. When they debug a complex nested query, they tend to divide it to simpler components, which again can be time consuming. The participants stressed that DBMS errors and execution plan are hard to interpret, and provide little help understanding the errors.

The participants found the following two techniques to be useful to mitigate these problems:

* **Using visual aids** – many of the participants believe visual aids is for debugging. 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 they could benefit that summary statistics 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.

As a summary it seems that a useful tool, will provide visualization of logical and execution plan and provide indicators of metrics on top of it.

##### 2.7 Multi Query Optimization

In addition to better visualizing a single query, it turns out there is not visualization technique for multiple queries. Multi-query optimization focusses on the optimizer itself. Traditional query optimizers are not appropriate for optimizing queries with common sub-expressions, since they make locally optimal choices, and may miss globally optimal plans as the following example demonstrates. Multi-query optimization (MQO) aims to find similarities among a set of queries and uses a variety of techniques to avoid redundant work during query execution. For database systems, MQO trades some small optimization overheads for increased query performance, using techniques such as “exhaustive search” on the search space [11], or sharing sub-expressions [12].

12 **<https://dl.acm.org/doi/abs/10.1145/342009.335419>**

13 **<https://dl.acm.org/doi/abs/10.1145/1516360.1516376>**

14 **<https://dl.acm.org/doi/abs/10.1145/42201.42203>**

##### 2.8 Sankey Diagrams

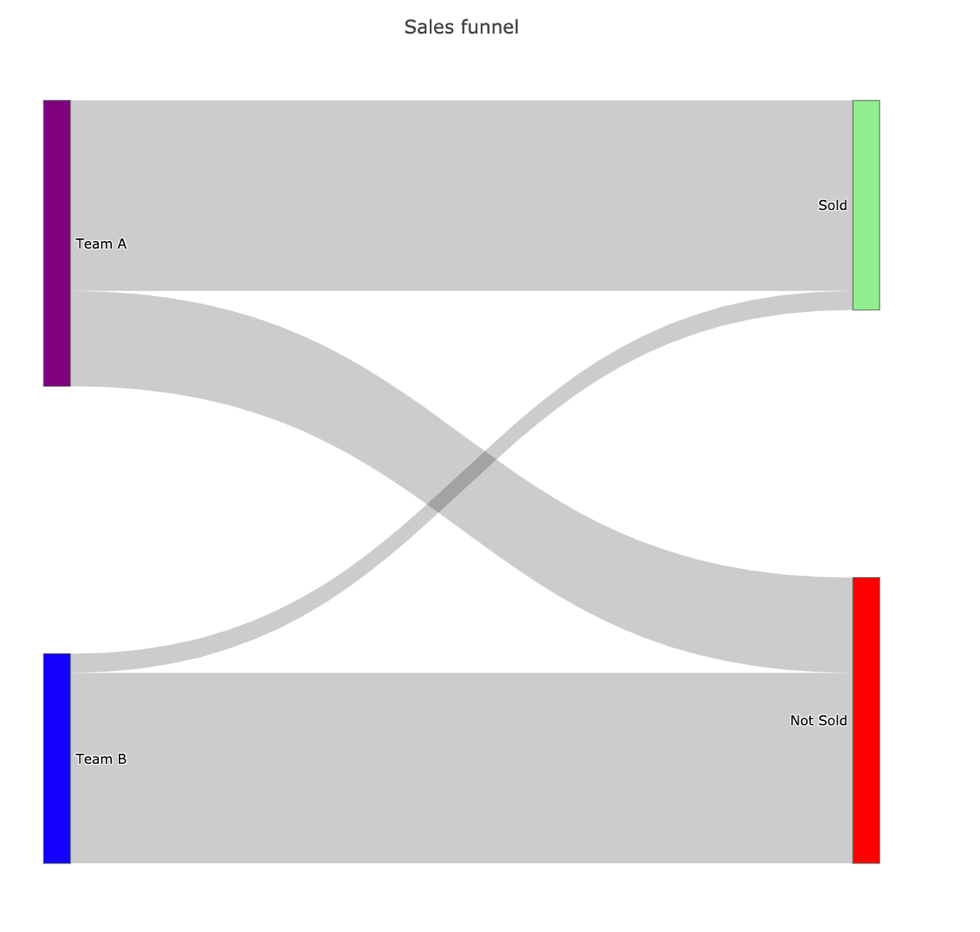
As we described in the previous sections, a tool with capabilities to visualize both the logical and execution structure is useful for DBMS’ users. Although most of these visualization techniques use graphs to represents the query sub-expression, they often lack the ability to understand the subexpression characteristics.

The reason most of these visualizations lack the ability to understand the subexpression characteristics is simple. They don’t use a visualization technique that emphasizes the difference in a specific measurement incomparable way while keeping the subexpressions hierarchy clear. Sankey-diagram is a visualization technique that can mitigate it. As it allows displaying any kind of measurable flow.

Sankey-diagram is a graph representation with specific characteristics. 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**

TODO effectiveness of paper

Save another 2 figures

**Figure 10**

**Figure 11**

##### 2.9 Summary

In this section we have shown the approaches to identify and mitigate flaws in SQL queries. We showed that out of those techniques visualizing the execution plans seems to be the most promising, but still lacks, and we have shown Sankey-diagrams’ merits and how they can mitigate it.

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 to provide a performant solution called 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. In Sankey, several nodes are represented by rectangles, their edges are represented with arrows that have a width proportional to the importance of the flow. 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.

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.

3.1 QueryFlow design  
In this section, we give an overview of QueryFlow’s design and how it is built. 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 plan or the query execution plan. QueryFlow process includes the following steps (Figure 12):

1. **QueryFlow parsing**- a SQL query is given as the input of QueryFlow. To transform a SQL query to its logical plan or execution plan, we modify the query by adding either *EXPLAIN ANALYZE* or *EXPLAIN* clause. The optimizer translates the query to its logical or execution plan and returns it as a *JSON* or a *YAML*. Then we will start QueryFlow’s parsing phase, which takes the raw execution plan and transform it into a structured representation to make it easier to work with and to support finding flaws across multiple queries.
2. **QueryFlow enrichment –** The structured representation by itself is useful, 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.
3. **QueryFlow visualization**- We start the visualization by transforming the enriched structured representation into a representation that is more suitable for Sankey-diagrams. We then visualize the enriched structured representation using Sankey diagrams. This allows emphasizing the important query’s characteristics and statistics.

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, only the parsing of the subexpression needs to be implemented.

##### 3.2 QueryFlow Parsing

The parsing stage begins with executing the queries to get the execution plans. To get the execution plan we use either the *EXPLAIN* or the *EXPLAIN ANALYSE* clause*.* The difference between *EXPLAIN* and *EXPLAIN ANALYSE* is that the first only give us estimated statistics about the query and the second actually executes the query and returns the real statistics. This will provide us with an execution plan with relevant and useful statistics of each sub-expression. There are various statistics we receive from the execution plan, like the number of records the sub-expressions hold (or estimation), or the total cost of the subexpression (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, 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 add certain 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 is useful as is, 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 got a parsed execution plan. Unfortunately, lacks some important statistics, and some statistics are not in the right granularity. To mitigate it, we are going to infer extra in the enrichment phase.

It’s important to know that the enrichment phase is valuable for both enriching the planned execution plan and the actual execution plan. We are going to infer new statistics from the existing ones, some of the more prominent missing statistics 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 subexpression 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 subexpression with the highest duration. Unfortunately, we only get the *total\_time* until the subexpression 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 and 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 subexpression 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 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 subexpression graph:

Text

Description automatically generated

**Figure 13**

In the next section, we will how QueryFlow’s visualization phase take the enriched execution plan and provide intuitive way to identify flaws.

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 parse 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. 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 edge. The first set of heuristics 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 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

In the next section, we provide a detailed example of QueryFlow execution.

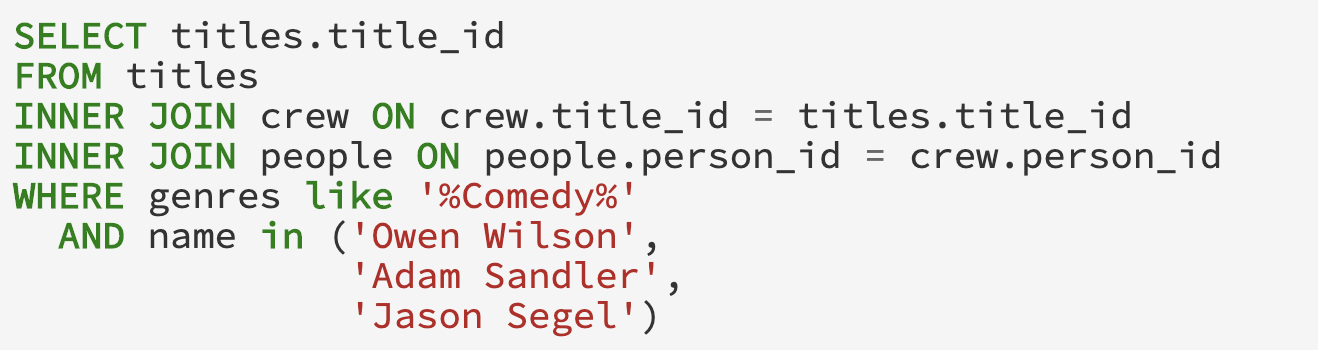
##### 3.6 QueryFlow Detailed Example

Through chapter three and chapter four, 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 query as the input 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.



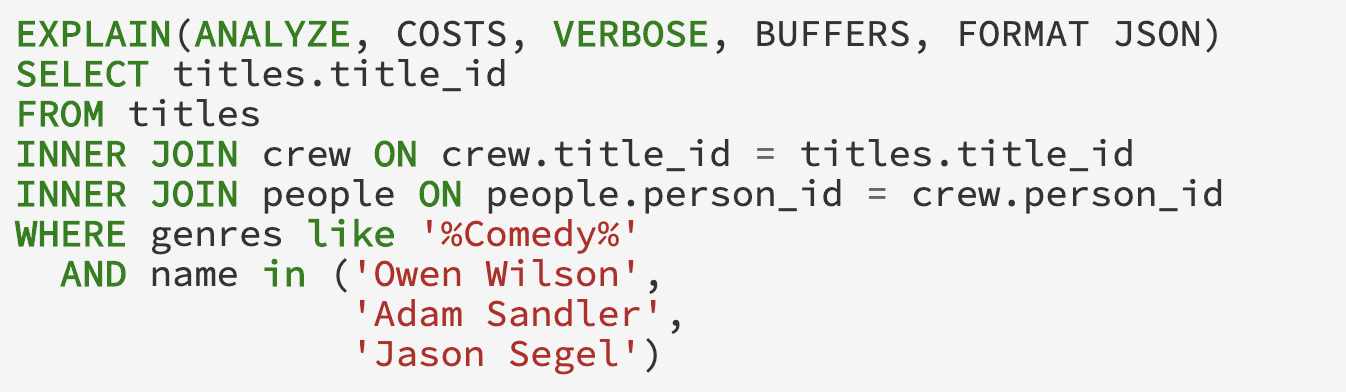
**Figure 15**

After we are given the query in Figure 6, 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 planned execution plan or the actual execution plan (the query is executed) depending on the input. In this example, I will use *"EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)” in* order to get execute the query, get accurate statistics, and adding metrics on buffers in a *JSON* format. If we omit the *ANALYZE* keyword, we will get the planned execution plan (the query is not modified). The modified query that returns the planned execution plan can be seen in Figure 16, and the query that returns the actual execution plan can be seen in Figure 17.

Text

Description automatically generated

**Figure 16**



**Figure 17**

When executing the modified query in Figure 16 and Figure 17. The results will be in a nested *JSON* format in order to make parsing easier and will include more statistics than default behavior due to the *COSTS*, *VERBOSE* and, *BUFFERS* parameters.  
  
Both execution plans are a huge *JSON*  that includes a lot of information, that is not important to understanding the gist QueryFlow. Thus, for readability purposes, we show the truncated *JSON* representation of the actual execution plan using Figure 18. A picture containing text

Description automatically generated  
 **Figure 18**   
  
Just for the sake of comparison, the planned 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 planned and actual execution plan are a nested *JSON* which represents the relational subexpressions. Each subexpression has different keys depending on the type of the subexpression and whether it’s the planned execution plan or the actual execution plan. These subexpressions describe the nature of part of the query and how it was (or going to be) executed. Each sub-expression has a lot 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 subexpressions.
* **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.*

A more intuitive way to think about both the planned execution plan and the actual execution plan is as a tree structure. From this step forward, it is the same for both the planned execution plan and the actual execution time so I will demonstrate the latter. 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 subexpression 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. In order to emphasis the parsing capabilities of removing irrelevant subexpression, 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 hierarch:

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, our query execution is clearer. We have separated *Scan* sub-expressions into two, one for the *Scan* and one for the filter. In addition, as we are looking for flaws in cardinality, we can drop subexpressions that don’t change the cardinality like the *Hash* subexpression 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 using the algorithm in Figure 13.   
  
The algorithm in Figure 13 takes the table from Figure 18 and iterates the source and target 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 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 subexpression cardinality.
* **actual\_duration –** is one of the metrics we want to measure that represents the subexpression 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 14 represents each parsed subexpression as a row. In our example the subexpressions rows can be described as follows:

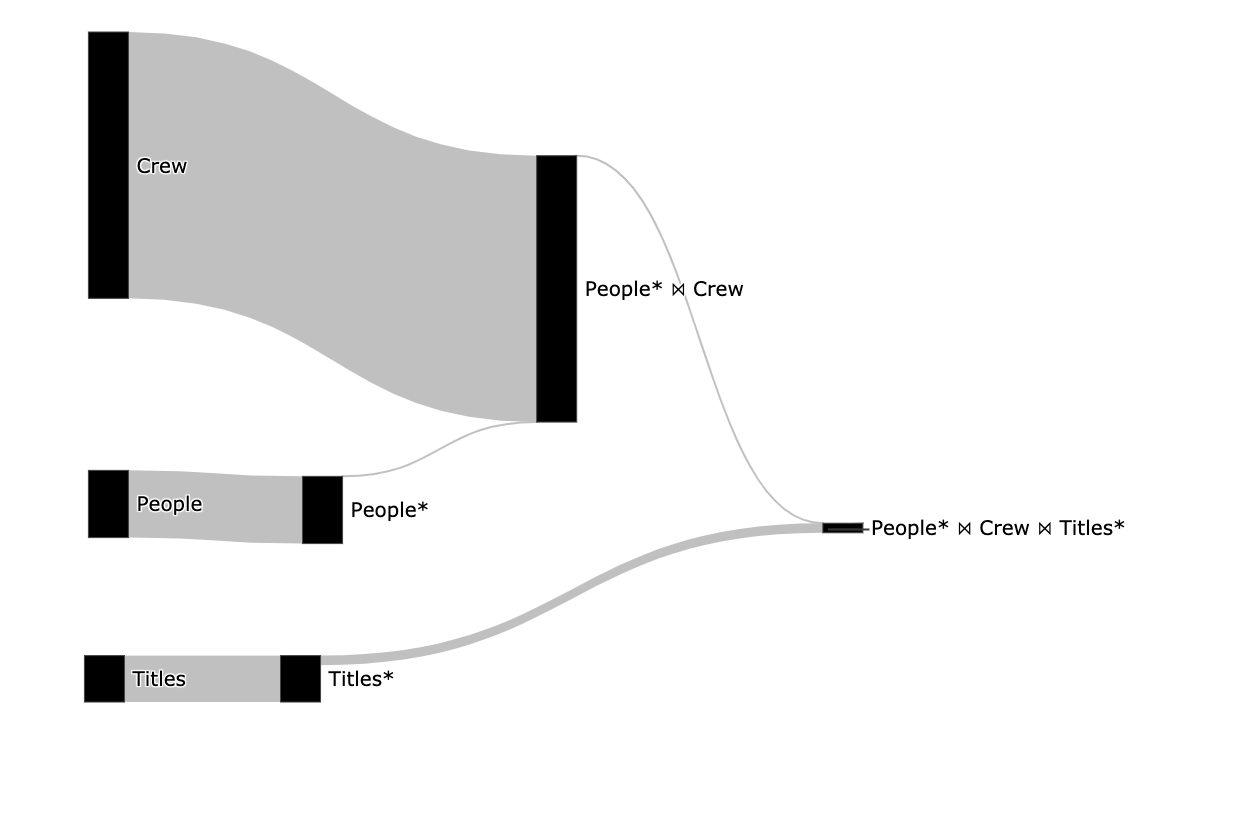
1. The node representing *People* is a table scan with 3,446,261 rows and has an edge to *People\**.
2. The node representing *People\** has 3 rows and has an edge to *People\* ⋈ Crew*.
3. The node representing *Crew* has 13,651,901 rows and has an edge to *People\* ⋈ Crew*.
4. The node representing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

Since none of the coloring heuristics are met will stick to the default coloring scheme and we will have gray edges and black nodes.

We are going to plotour table representation to visualize our query as Sankey-diagram. The nodes will subexpression label. While edges will represent a parent-child relationship. The width of the edges will be proportional to the cardinality (*actual\_rows)*, and additional details regarding each subexpression are provided when hovering over an edge.



**Figure 24**

After we created the visualization, we can see the subexpression hierarchy and the cardinality of each subexpression, 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 subexpression 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 planned 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. QueryFlow excels in identifying 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*.
* Queries bottlenecks – finding the queries bottlenecks, for a single query or for multiple queries.
* Optimizer problems – by comparing the optimizer expectations and reality for subexpressions.

The rest of this chapter is structured as follows:

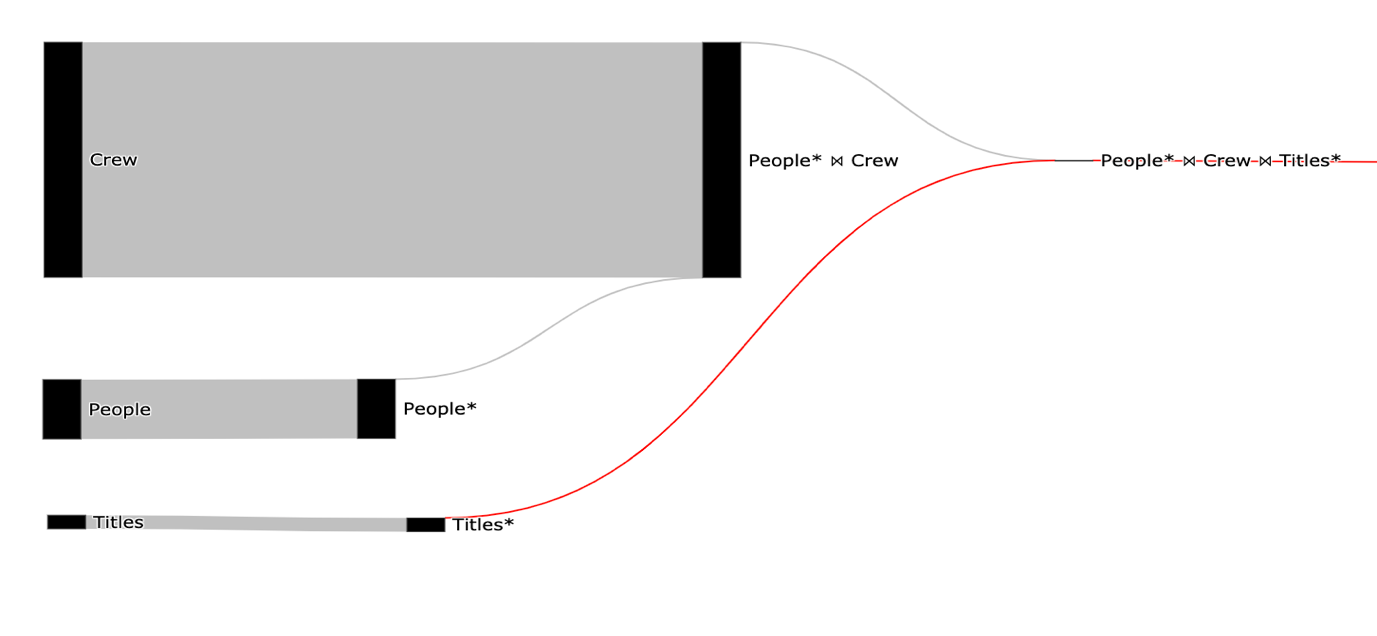
* Section 4.1 provides an example of identifying missing records using QueryFlow.
* Section 4.2 provides an example of identifying duplicated entries using QueryFlow.
* Section 4.3 Identifying performance bottlenecks in a single query using QueryFlow.
* Section 4.4 provides an example of identifying 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. By using QueryFlow to visualize the cardinality (*actual\_rows)* of the sub-expression of our query we can find missing records 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 to return an empty result. The query can be seen in Figure 25.

****

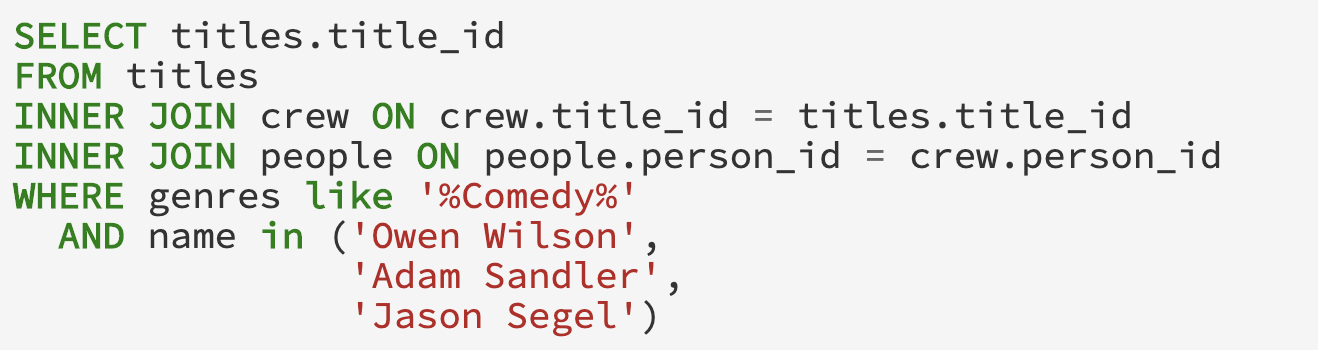
­­  **Figure 25**  
We are going to use QueryFlow to identify the empty results (empty result will have red edge) and where it was originated. The corresponding Sankey that represent the cardinality sub-expression of our example can be seen in Figure 26.

****

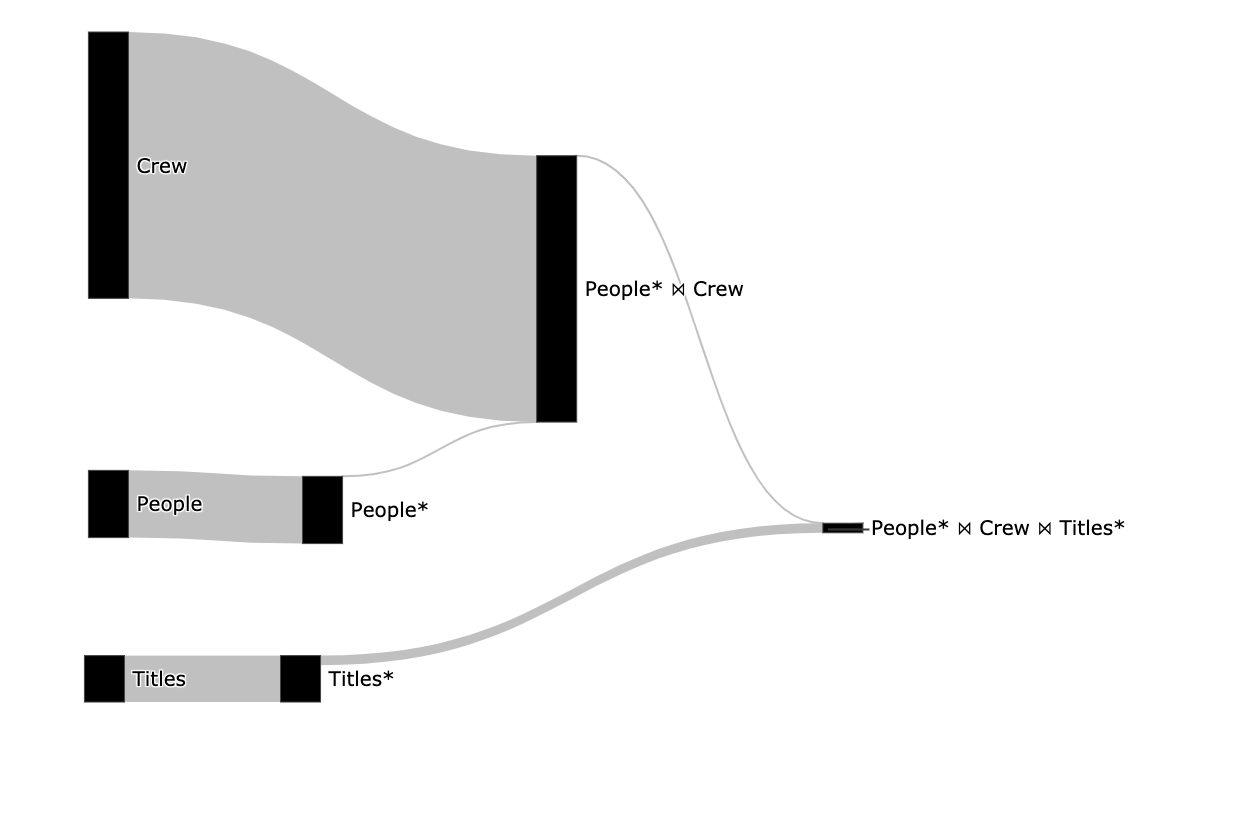
­­ **Figure 26**

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 return empty results is that there is no lower-case *comedy* value. We will rewrite the predicate to be Camel-Case and we will support multiple genres in the same movie. The fixed SQL query can be seen in Figure 27.



­­ **Figure 27**  
To validate that our query is actually fixed, we will visualize it again. The corresponding Sankey that represents the cardinality of our modified query can be seen in Figure 28.



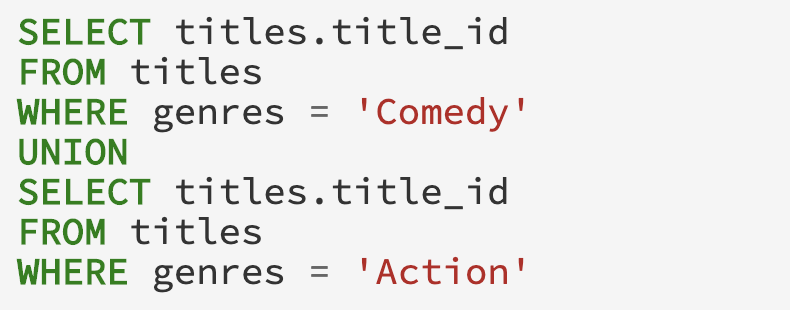
­­ **Figure 28**

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

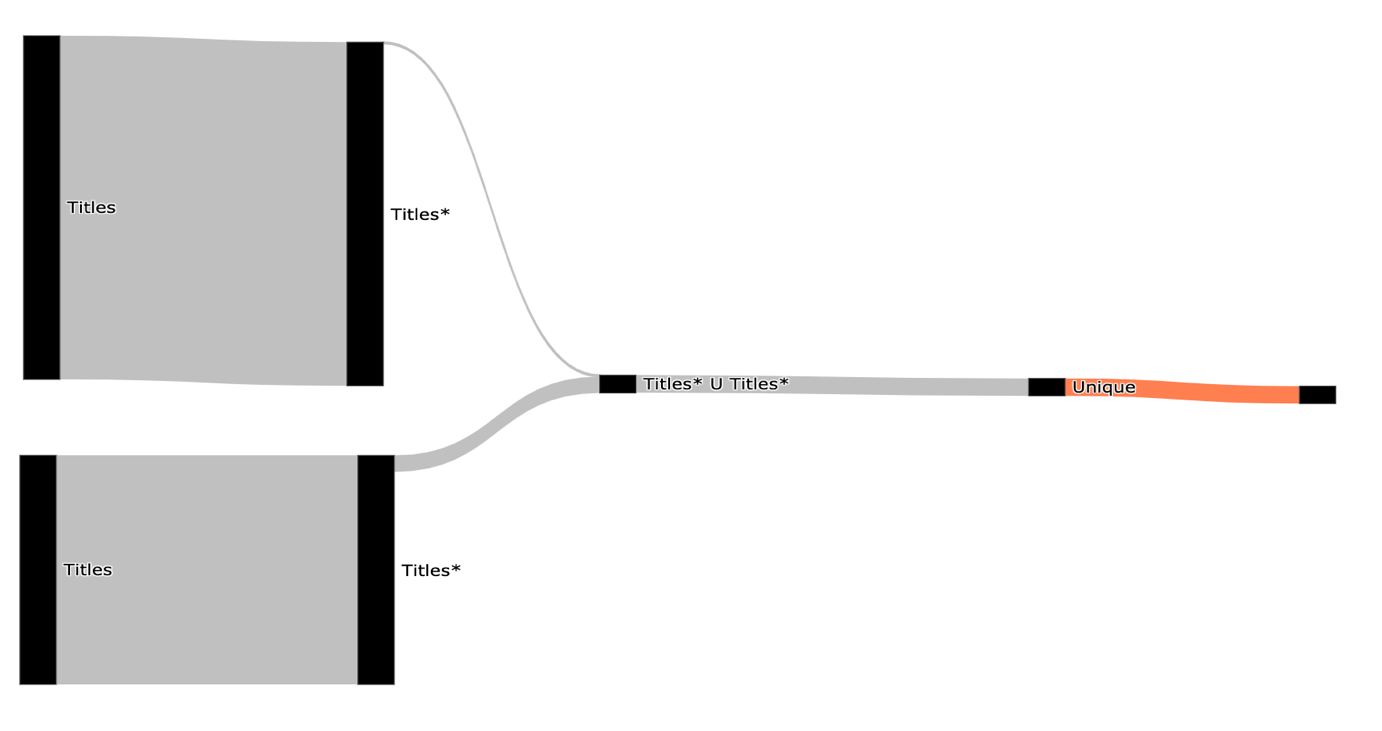
* We fixed the empty result set (there are no red edges).
* We understand the size of the relation, for example *Crew* is much bigger than people and *Titles.*
* The filter on both *People* and *Titles* relations is effective and from performance stand of point can enjoy optimization techniques like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

##### 4.2 Identifying Ineffective Operations

Problems related to infective operation are really common and finding them tends to be really hard for non-experts. Although these operations won’t make your query result wrong it will slow it down. By using QueryFlow to visualize the cardinality (*actual\_rows)* of the sub-expression of our query we can find ineffective operations due to the *DISTINCT*, *UNION*, *WHERE,* and *HAVING* clauses.

We introduce a simple example, that is rich enough to illustrate the gist of the QueryFlow. Consider the following question, “find 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) and where it was originated. The corresponding Sankey that represent the cardinality of our example can be seen in Figure 30.

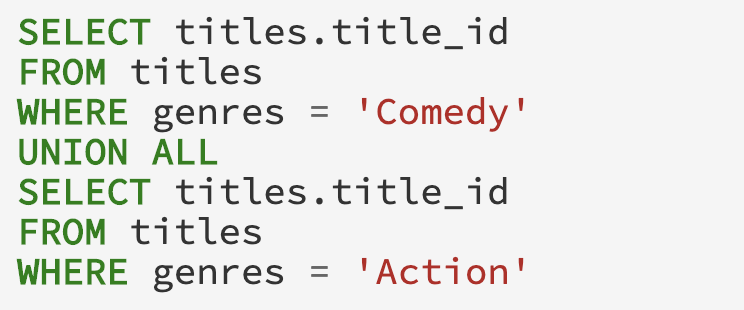


­­ **Figure 30**

Using the Sankey visualization, we can clearly see that the *Unique* sub-expression is redundant as it marked in orange or by hovering both operations and looking at the number of rows.

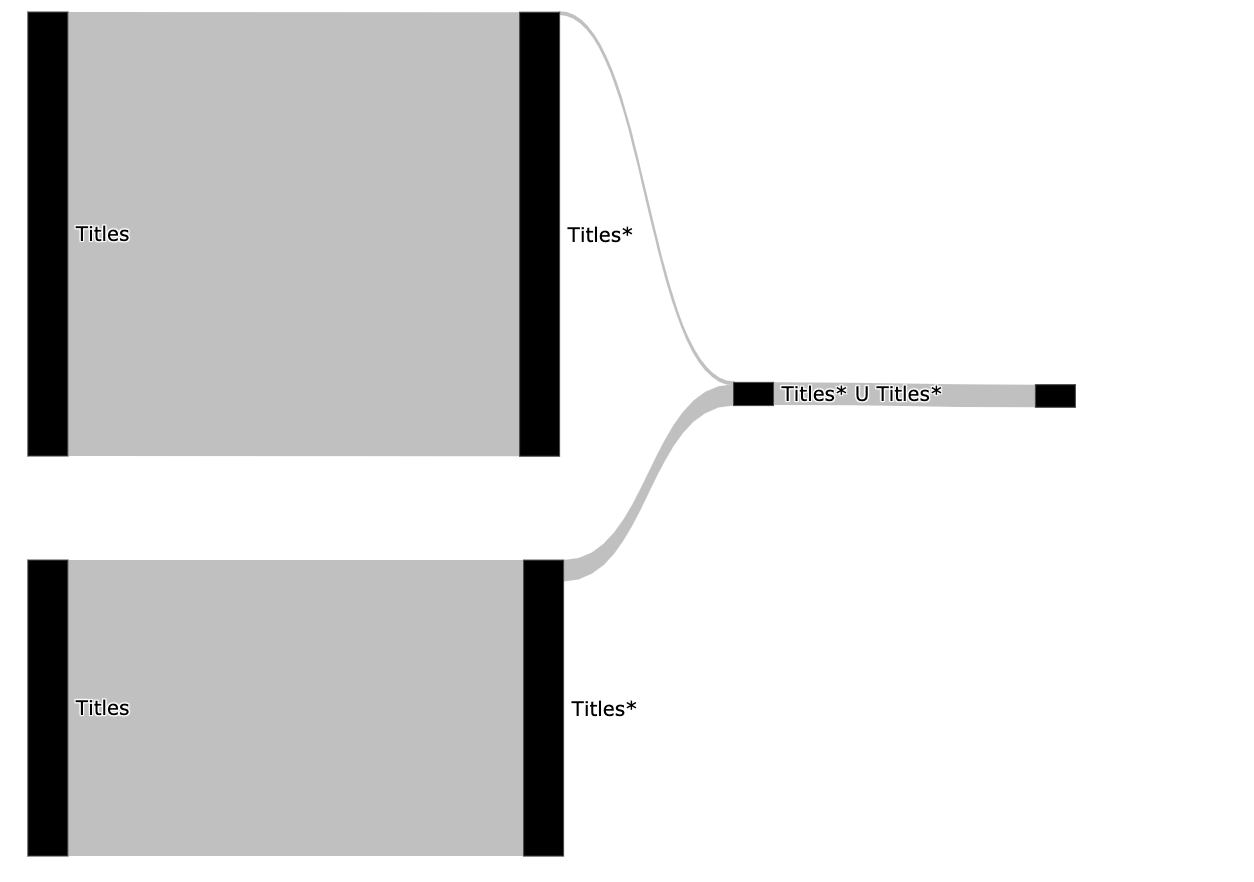
In PostgreSQL the difference between *UNION ALL* and *UNION* is that the first only append two relations and the second remove duplication after the append. Since both relations are disjoint, there is no need for removing duplications after the append. This will not affect the query correctness but will 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 actually fixed, we will visualize it again. The corresponding Sankey that represents the cardinality of our modified query can be seen in Figure 32.



­­ **Figure 32**

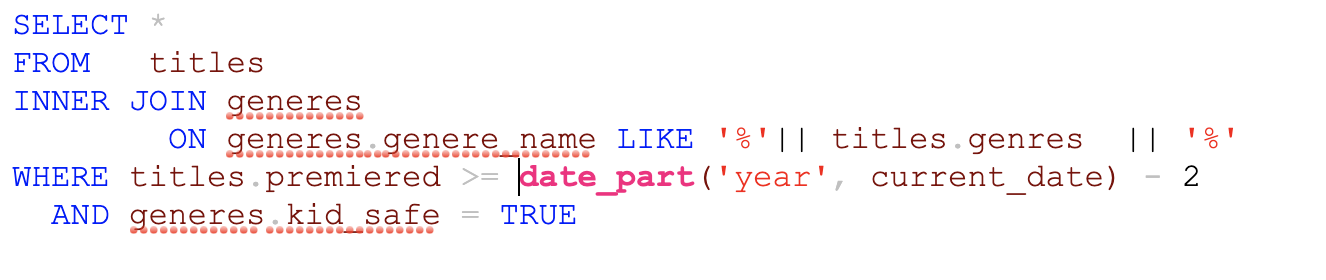
From this simple visualization in Figure 32 we can infer the following:

* 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.

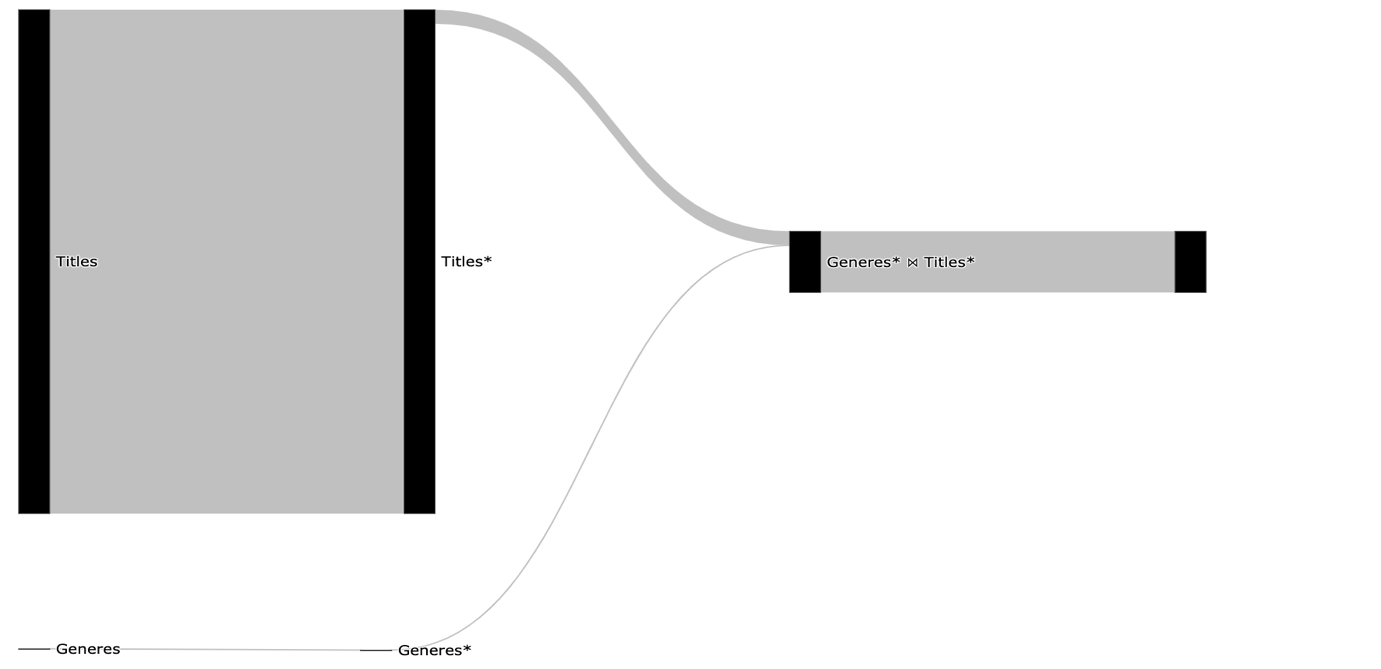
##### 4.3 Identifying Duplications

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 ineffective operations due to the *JOIN*, *UNION ALL* clauses.

introduce a simple example, that is rich enough to illustrate the gist of the QueryFlow. Consider the following question, “find all movies with genres that are safe for kids from the last two years”. This question is equivalent to the SQL query in Figure 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 visualize it again. The corresponding Sankey that represents the cardinality of our modified query can be seen in Figure 36.

Diagram

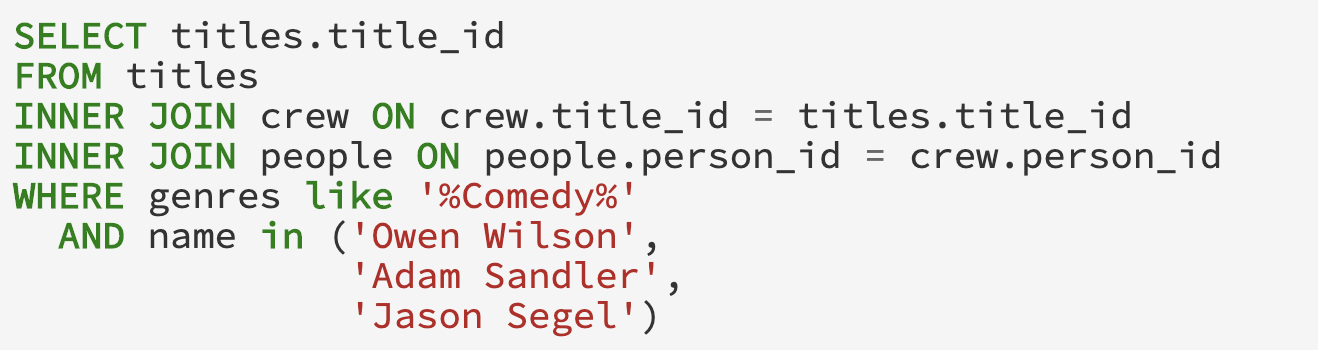
Description automatically generated

**Figure 36**

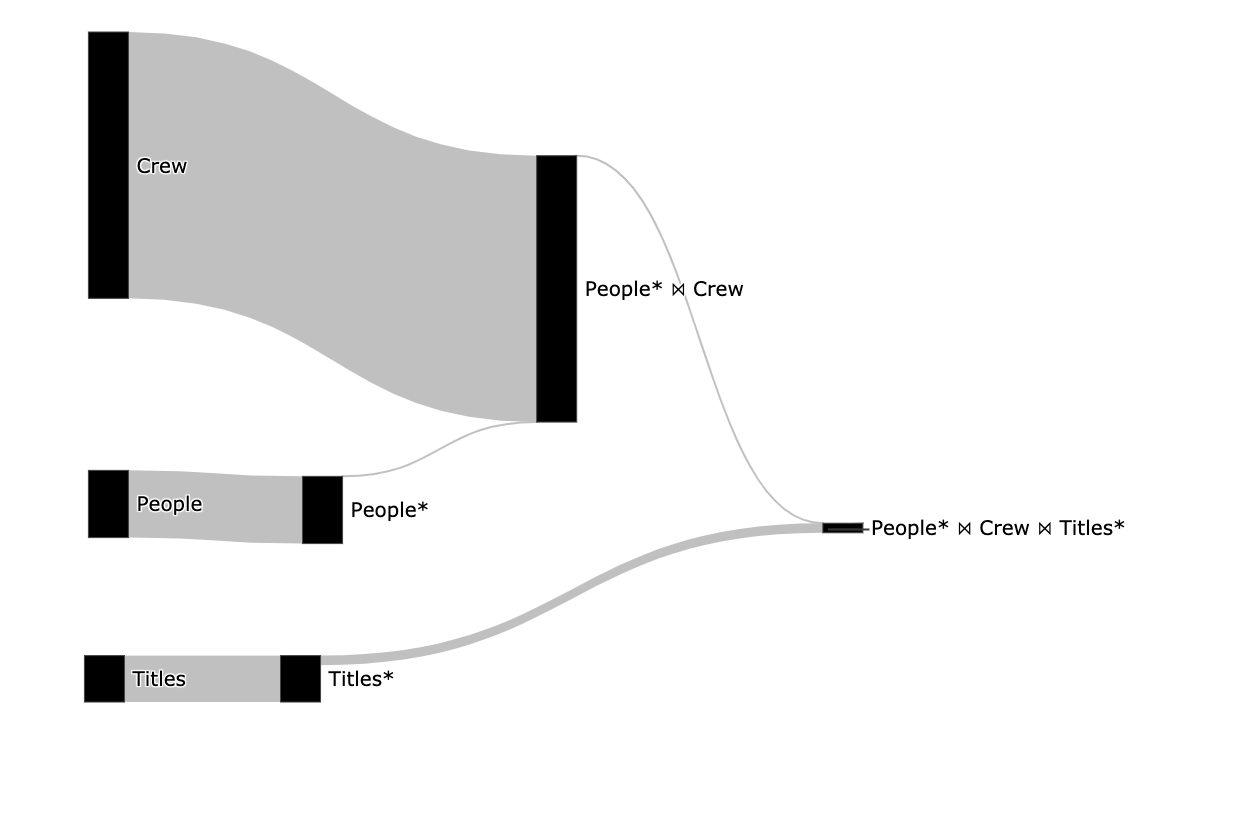
We can immediately see (Figure 36) the join still introduces the duplications but after the *Sub query* clause, we are removing those duplications in the *t\** sub-expression.

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

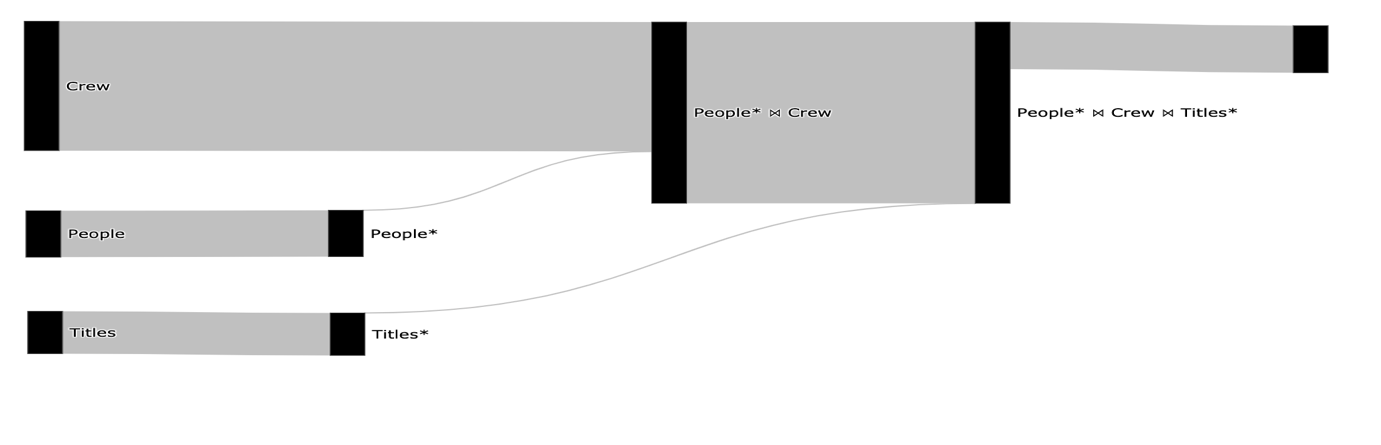
Slow queries are really common and finding them tends to be really hard for non-experts. By using QueryFlow to visualize both the cardinality (*actual\_rows*) and duration (*actual\_duration*) of the sub-expression of our query we can find the bottleneck in the query.

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 of our example 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.

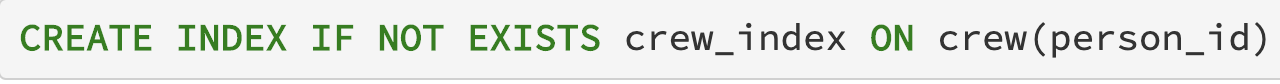


­­­­ **Figure 39**

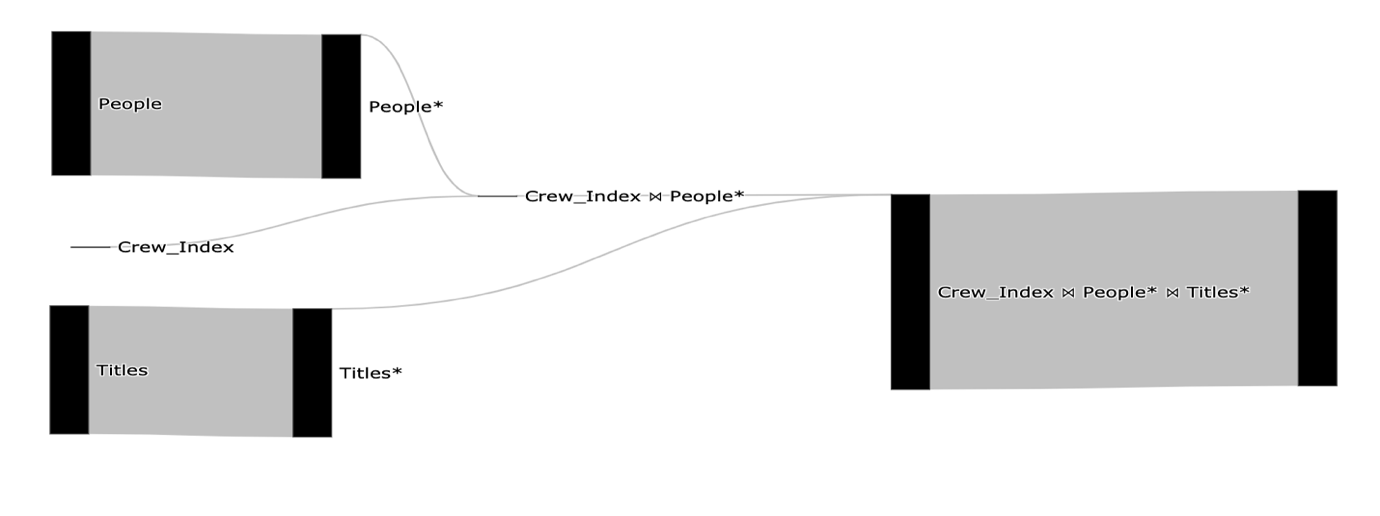
The total query duration is 7 seconds, we can see in Figure 39 that the longest operation (3.6 seconds) is the People\* ⋈ Crew.

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 visualize it again. The corresponding Sankey that represents the duration of our query now can be seen in Figure 41.

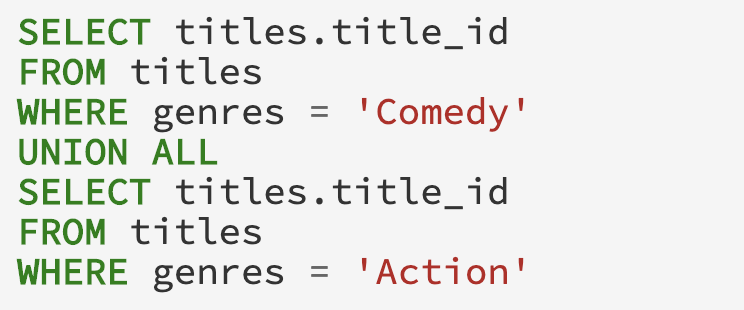
****

­­ **Figure 41**

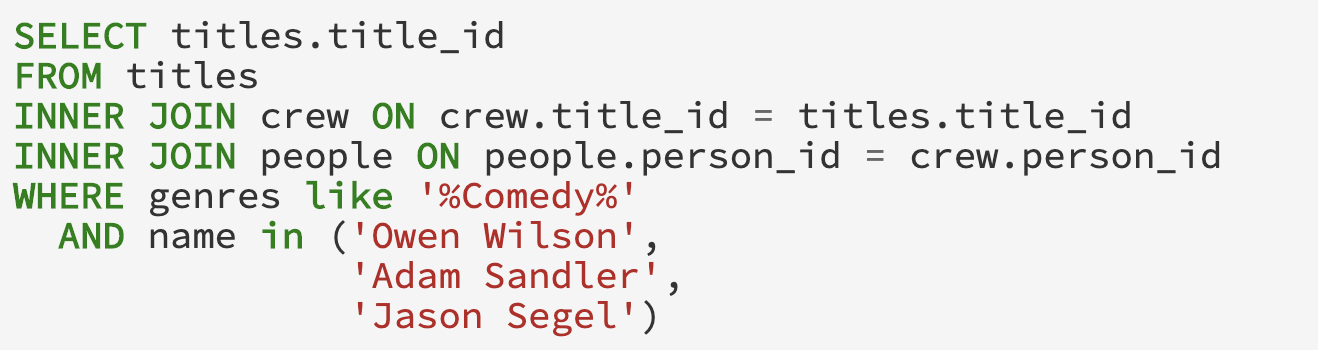
The total query duration is 2 seconds, we can see in Figure 41 that it improved both the *People\* ⋈ Crew* and the scan on the *Crew* relation.

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 finding them tends to be really hard for non-experts. By using QueryFlow to visualize both the cardinality (*actual\_rows*) and duration (*actual\_duration*) of the sub-expression of our queries we can find the bottlenecks in our queries. The main reason multi queries optimization is hard, is that one query may affect the other. Thus, it is critical to make sure that take other queries into consideration

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

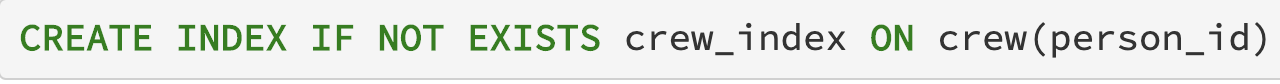
Description automatically generated­­

**Figure 45**

The total query duration of both queries is 46.5 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 visualize it again. The corresponding Sankey that represents the duration of our query now can be seen in Figure 47.

­­ **TODO Figure 47**

The total query duration is X 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 willcontinue to improve our queries. We see that both queries uses*Title* scan and in particular have predicate over *comedy*. We will add a partial index on the Titles relation using the title\_id column on only Comedy movies.   
  
The index creation query on the crew relation with the person\_id column can be seen in Figure 48.   
­­ **TODO Figure 48**  
To validate that we eliminated the bottleneck, we will visualize it again. The corresponding Sankey that represents the duration of our query now can be seen in Figure 49.

­­ **TODO Figure 49**

The total query duration is X seconds, we can see in Figure 49 that it improved both the *People\* ⋈ Crew* and the scan on the *Crew* 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 itself are really common and finding them tends to be really hard for non-experts. By using QueryFlow to visualize both the actual and estimated cardinality (*plan\_rows, actual\_rows*) and the actual and estimated duration (*cost, actual\_duration*) and of the sub-expression of our query we can stale statistics and relations that need to be vacuum.   
  
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 visualize it 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* subexpression is pretty proportional.

##### 4.7 When QueryFlow won’t help

In the chapter covered use cases and flaws QueryFlow can identify. Like all tools, QueryFlow won’t be effective in all the cases like the famous quote says “There is no free lunch”.   
  
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 visualize the same metric using percentage, this will fix this issue 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 as can be seen in Figure 56.

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.

# 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. 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. 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.

Section 5.1 provides an overview of the setup. Section 5.2 evaluate possible optimizations that don’t require query modification. Section 5.3 evaluate possible optimizations that require query modification. Section 5.4 provide an overview of lessons that can be learned for SQL best practices.

##### 5.1 Setup

All the experiments were performed on a single machine with 32 GB memory and 8 cores. Optimization times are measured as CPU time (user + system). 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 2 - Consists of the base row size x 10 and is 10 GB in size

We executed each benchmark several times in order to provide a more comprehensive evaluation.

##### 5.2 Optimization without Queries Modification

* Q6,Q12 will contribute parallel index scan to improve execution time from aggregate to partial aggregate
* ⁃ Q13 improved by x5 from parralal index  shared\_buffer/effective\_cache\_size =10GB  work\_mem = 1GB

##### 5.3 Optimization with Queries Modification

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

CP1.2: Interesting Orders.

CP2.1: Large Joins.

Q20 Flattening Subqueries.

CP5.2: Moving Predicates into a Subquery. 17/16 21

CP 4.3a: Rewrite LIKE(X%) into Range Query. 16,21

# 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. 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] 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). ACM, New York,

NY, USA, 1095–1098. DOI:

http://dx.doi.org/10.1145/2588555.25945

[3] 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).

[4] 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)

[5] [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)

[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] [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)

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

[9] [J. R. Haritsa. The Picasso database query optimizer visualizer. PVLDB, 3(2):1517–1520, 2010.](http://vldbarc.org/pvldb/vldb2010/pvldb_vol3/D01.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] <https://www.cs.umd.edu/~leilani/static/papers/gathani_debugging_CHI_2020.pdf>

[12]

[13]

[14]

[15] P. Riehmann, M. Hanfler, and B. Froehlich. Interactive Sankey diagrams. In Proc. IEEE Symp. Information Visualization, pages 233–240, 2005.

Timos K. Sellis. Multiple query optimization. ACM Transactions on Database Systems, 13:23–52, March 1988.  
P. S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In Proc. SIGMOD, 2000.