**The Open University of Israel**

**Department of Mathematics and Computer Science**

**Thesis Proposal**

**Visualizing Database Execution Plans using Sankey**

A 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**

ID: 200953859

050-8768065, [eyaltrabelsi@gmail.com](mailto:adi.swissa123@gmail.com)

28 Mevo Walker, Tel Aviv

Prepared under the supervision of Prof. Ehud Gudes

2020

**Personal Information:**

Eyal Trabelsi, id: 200953859

28 Mevo Walker St., Tel Aviv

Phone: 050-8768065, Mail: eyaltrabelsi@gmail.com

**Thesis Name**

Visualizing Database Execution Plans using Sankey.

**Advisor Name**

Prof. Ehud Gudes – Department of Mathematics and Computer Science, Open University, Israel

**Thesis Goal**

Database management systems (or DBMS) have been around for decades and yet are still difficult to use, particularly when trying to identify flaws in user queries. Since it’s rare for users to write queries without any errors or any performance issues, identifying and understanding those errors and bottlenecks can improve productivity tremendously and aid in making DBMS easier to use.

In this work, I am focusing on how we can identify those flaws, by representing the query’s execution plan in an intuitive way using Sankey diagrams. This gives the users an intuitive understanding of the query characteristics, by observing how the query is executed under the hood.

**Proposed Work**

A new query visualization using Sankey-diagram, that allows finding flaws in Databases. The rest of this article structured as followed:

* Chapter 1 – Introduction to identifying flaws in Databases.
  + Section 1: Motivation of QueryFlow
  + Section 2: Related works for identifying flaws in databases queries.
* Chapter 2 –QueryFlow Design.
  + Section 1: QueryFlow design in general.
  + Section 2: QueryFlow parsing.
  + Section 3: QueryFlow enrichment.
  + Section 4: QueryFlow visualization.
  + Section 5: QueryFlow detailed example.
* Chapter 3 – QueryFlow use-cases
  + Section 1: Identifying missing records.
  + Section 2: Identifying Ineffective operations.
  + Section 3: Identifying duplications.
  + Section 4: Identifying performance bottlenecks in a single query.
  + Section 5: Identifying performance bottlenecks in multiple queries.
  + Section 6: Identify flaws in the optimizer itself.
  + Section 7: When QueryFlow won’t help.
* Chapter 4 – Evaluation of QueryFlow on TPC-H benchmark.

**Main contributions**

1. **Query Execution Plan Enrichment –** Databases tend to show raw statistics for queries. It can be useful to infer and enrich the query execution, some of the enriched statistics:
   * **Subexpression missing statistics-** some of the statistics are not in the right granularity. For example, the *Actual Total Time* is cumulative and include the ancestor sub-expressions, so we would like to fix the statistics to be in the sub-expression granularity.
   * **Percentage statistics** - representation of certain characteristics as a percent. This becomes critical when we compare two metrics measured by different metrics like *Total Cost* and *Actual Total Time.*
   * **Redundant operations**- whether an operation is effective or not, for example, a non-effective operation will be a *distinct* operation that filters nothing.
2. **Using Sanky-Diagrams to Visualize Flaws in SQL Queries -** A new representation for queries as a Sankey diagram that allows us to understand the nature of a query or multiple queries (This is the first technique to visualize in compact manner multiple queries).  
   It can be used to find the following:
   * Cardinality issues- due to WHERE*, JOIN, UNION, DISTINCT, HAVING* clauses. For example, let’s say we are looking to find the most similar products and we are getting duplicates in the results it can help us to pinpoint that the issue is in our join condition which may be wrong.
   * Queries bottlenecks - For example, let’s say we are looking to find the most similar products, here, the heaviest part is the join, it is simple to find it using the Sankey diagram, and in addition, it will give information that allows understanding whether a better join strategy or more refreshed statistics may help.
3. **Using Sanky-diagrams to visualizing flaws in the Optimizer -** A new representation for queries as a Sankey diagram that allows us to understand the nature of a query as opposed to what the optimizer would expect. For example, let’s say we are looking to find the most similar products, here, the optimizer expects the heaviest part is the join according to cost statistics, but in reality, the bottleneck is not there according to the actual execution time, it will give information that allows understanding whether it’s an issue with stale statistics or with the optimizer configuration.

**Chapter 1: Motivation and Related work**

**1.1 Motivation for QueryFlow**

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 debugging of queries is a necessary step towards learning to use DBMSs effectively.  
  
Debugging a complex query and bring it to perfection is challenging. Often it requires more than simply fixing syntax issues, as query might return unexpected results like zero entries, duplicate entries, incorrect results, or not meet the performance requirements. Due to SQL declarative nature, the translation from a query to an execution plan is difficult for most users to identify what is the problems and to provide a performant solution.

One of the techniques to debug DBMS queries is to utilize and visualize the execution plan statistics. This visualization gives the users a better understanding of the query characteristics, by observing how the query is executed under the hood. The execution plan provides characteristics and statistics on each intermediate result of the sub-expression it allows us to understand and track how the data “flow” in each sub-expression of the query.   
  
In this paper, we contribute QueryFlow, a query visualization tool that provides insights into common problems, such as performance bottlenecks and cardinality issues. 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.

**1.2 Related works for identifying flaws in Databases.**

In this section, we review the relevant literature in debugging and understanding query behavior.

The first strategy is to find issues in the data itself, this is done by analyzing the database query log (the history of queries that run on our database). Analyzing the queries that have modified the data in the past, and detecting which queries may have contributed to the errors. For example, we can use the query history to explain how errors occur in the database like QFix [1]. But it's will not provide a solution for issues in the actual query itself and thus quite limited.

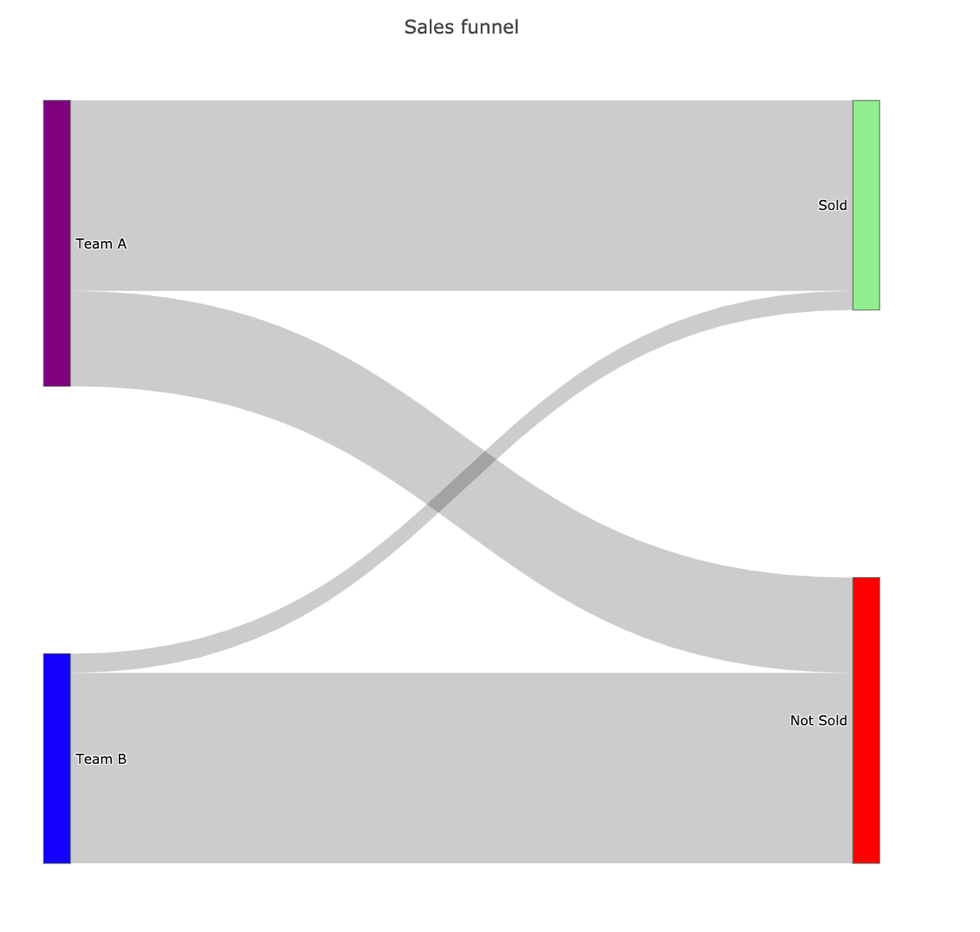
The second strategy is to provide debugging capabilities for the database. The first technique is the why-not systems [2] which explain why a specific input fails to appear in the result. The second technique is to add breakpoints capabilities to the query and retrieve the result of the sub-expression using views [3][4]. These debugging tools give the users a much more granular understanding of the DBMS. But they require either to know where the issues are or to use special systems. They are also, less intuitive than other visualization techniques and can consume much more resources than all other strategies.

The last and most prominent strategy is to infer query characteristics from either the logical or execution structure and to visualize it in an intuitive manner:

* The logical structure refers to the meaning of the operations within a query. The most common techniques illustrate the structure of a query via static analysis of the query (and avoid executing the query). For example, QueryViz [5] attempts to achieve exactly that like several others [6][7]. These techniques give an intuitive understanding of the query and by avoiding the execution of queries, these strategies scale to big data use cases. But this only provides a shallow understanding of the changes in the actual data.
* The execution structure refers to the intermediate results and statistics at various stages of query execution. The techniques illustrate the structure of a query via a dynamic analysis of the query execution and focus only on execution time. For example, Picasso [8] attempts to understand the queries' bottlenecks as well as others [9]. 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.

DBMS users can benefit from a tool with capabilities to visualize both the logical and execution structure. Many of these visualization techniques using a graph to represent the query subexpression, and Although it seems natural it lacks the ability to understand the subexpression characteristics. Sankey diagrams [10] are a visualization technique that can mitigate it. It allows displaying any kind of measurable flow. The idea of Sankey’s diagram is similar to a graph representation, where links are connected to nodes.   
- The nodes represent the entity and visualized as a colored rectangle.   
- The links represent a measurable metric and visualized as an edge with a width proportional to the metric measure.

Let’s say we have two sales teams and we want to compare their sales performance. In our example (Figure 1), we can clearly see that Team A sold a lot more than Team B as the width of the edge between “Team A” node and “Sold” edge is much thicker than the edge between “Team B” and “Sold”. Also, we can see that the conversion of Team A is better than Team B. because most of “Team A” node and “Sold” edge is thicker than “Team A” node and “Not Sold” edge as oppose to Team B.



**Figure 1**

The Sankey diagram allows us to show additional information if needed. We can show the value which represents our link width, our measurable metric, by simply hovering the link (Figure 2).

****

**Figure 2**

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].

QueryFlow will bring the following improvements to current tools:

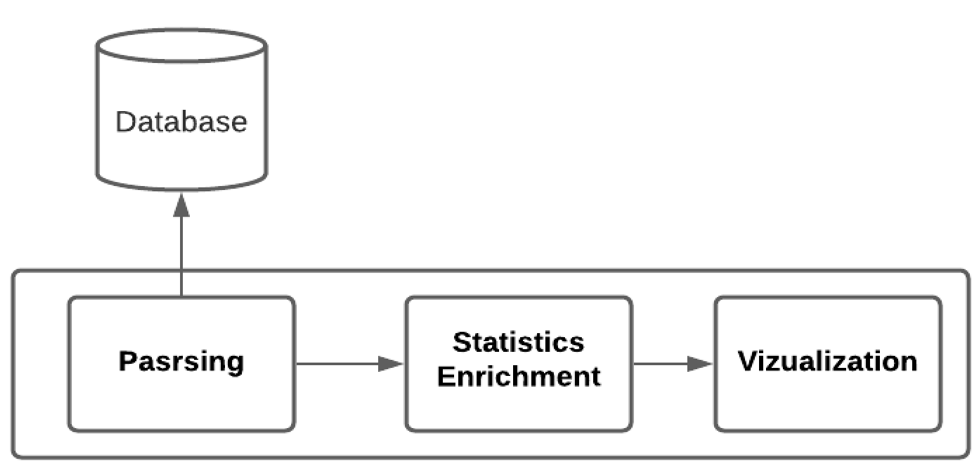
* Capabilities to visualize both the logical and execution structure.
* Represent query characteristics as part of the visualization.
* The ability to visualize multiple queries.

**Chapter 2: QueryFlow Design**

**2.1 QueryFlow design in general**

In this section, we review the design of QueryFlow and how it is built. We designed QueryFlow with minimal requirements for the database system. This makes it easier to generalize beyond one database, as it’s only requires a query execution plan. The process itself includes the following two steps (Figure 3):

* 1. **QueryFlow parsing**- Given a SQL query we are parsing the execution plan we got using *EXPLAIN ANALYZE* or *EXPLAIN* clause, in order to transform raw JSON / YAML to a structured representation.
  2. **QueryFlow enrichment –** The structured representation 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 additional statistics using the existing ones. This will give us an execution plan with much better statistics.
  3. **QueryFlow visualization**- Visualize the enriched structured representation using Sankey diagrams. This allows to emphasize the important query’s characteristics and statistics.

  
 **Figure 3**

In this article, 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 the subexpression needs to be implemented.  
 **2.2 QueryFlow Parsing**

The parsing stage is implemented using python and we begin with executing queries using SQLAlchemy to get the execution plan. We can get the execution plan by using either *EXPLAIN* or *EXPLAIN ANALYSE,* the difference between explain and explain analyse is that the first only give us estimated statistics about of the optimizer for the query and the second actually execute the query and return 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 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 to 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 subexpression 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 similar sub-expression to be represented by the same node (and have multiple edges). In order to do so we specify a hash function that indicate whether two sub-expressions from different queries are the same and we need to clear the cache between each.

When we parse the execution plan, we can add certain heuristics to have more compact representation of our query. For example, for some operations like *HASH* don’t have any effect on number of rows, and we can simply skip them when we parse the execution plan for cardinality purposes.  
  
  
**2.3 QueryFlow Enrichment**

For this reason, QueryFlow will enrich the parsed execution plan statistics. We are going to infer new statistics from the existing ones, some of the more prominent missing statistics are:

* **Subexpression missing statistics-** many statistics are cumulative and include the ancestor sub-expressions and we want it in a sub-expression level. For example, we want to know the duration of each-subexpression, the relevant statistics that we get is the *total\_time* until the subexpression (included), so we need to calculate the sub-expression duration by subtracting the total from the maximum *total\_time* of its ancestors.
* **Percentage statistics** – having certain statistics as percentage can be very useful. This becomes critical when we compare two metrics measured by different metric. For example, if we want to compare optimizer estimation and the actual execution time, we must compare them as percentage as they work with different units (not convertible).
* **Redundant operations**- calculating whether an operation is effective or not. For example, a non-effective operation will be a *distinct* operation that filters nothing, which can help us improve the query performance.
* **Human readable representation**- transform each subexpression label to be concise and understandable. For example, instead of representing a join like *T1 JOIN T2* we will present it as *T1 ⋈ T2.*

In order to enrich the statistics, I am using the following algorithm on the subexpression graph:

Graphical user interface, text, application

Description automatically generated  
 **Diagram 1**

**2.4 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. This makes Sankey perfect to visualize SQL queries. Since SQL queries have interesting statistics for their that we

intermediate steps such as cardinality and duration and to understand how the data “flow” in the query, and we construct those diagrams using Plotly.

Nodes will represent relations, each labeled according to the previous relations and operators applied to it and transform to a more intuitive form, for example instead of T1 JOIN T2 we will present it as T1 ⋈ T2. While edges, will represent a parent-child relationship, aka the operator that was applied to the relations and width proportional to the cardinality or duration, additional details regarding the operator are provided when hovering an edge.

In addition to that, I propose to add heuristics to emphasize which part of the Sankey   
diagrams might require special attention:

* When a Relation cardinality is zero.
* When an operation is redundant.   
  1. **QueryFlow Detailed Example**  
     Through chapter two and chapter three I am going to use the IMDB dataset. The dataset contains total eleven tables, but we only use *titles, crew, people and genre* and can be described by the Diagram 2.Diagram

     Description automatically generated

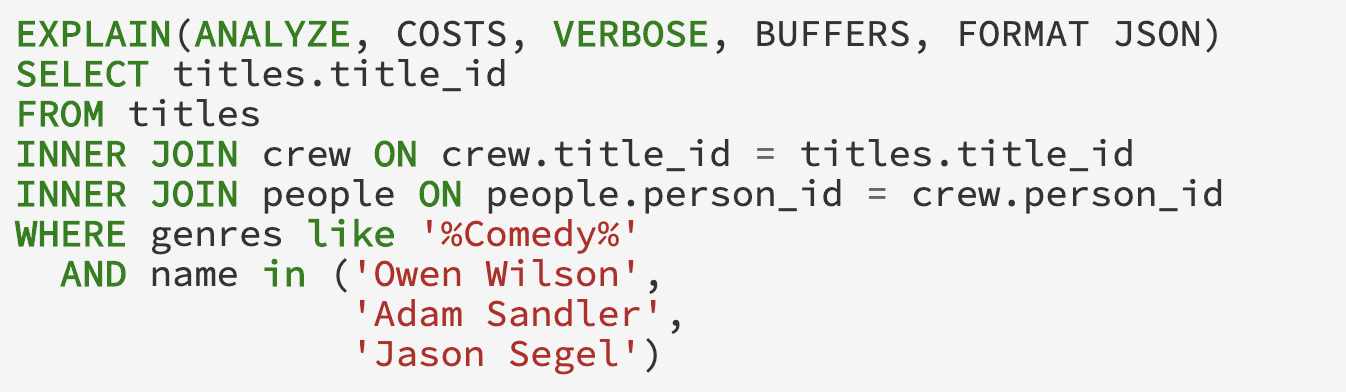
**Diagram 2**

In this example, the question I want to answer is “What movies are recommended for me, given that I love comedy movies with Owen Wilson, Adam Sandler, or Jason Segel” which can be described in the following way in SQL query in Figure 4.



­­ **Figure 4**

After we are given the query in Figure), we will modify the query by adding *EXPLAIN* prefix to the beginning of the query this will give us a new query which will return the actual execution plan. In this example I will use *"EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)” in* order to get execute the query, to get accurate statistics, and adding metrics on buffers in a JSON. The query that is actually executed by QueryFlow can be seen in Figure 5.



**Figure 5**

When executing the modified query in Figure 5, we will get the actual execution plan. The results will be in a nested JSON format in order to make parsing easier, and will include more statistics then EXPLAIN ANALYSE with the default behavior. Because the actual execution plan is a huge JSON and include a lot more metrics and information, I have showed Figure 6 only the importance elements in the JSON representation for this example.

  
 **Figure 6**   
  
As we can see the actual execution plan in Figure 6 is a nested JSON of operators. Each relational operation has different keys, which describe what the operation is and how it was executed, and the keys can be described by the following types of keys:

* **Node Type –** the type of operation it is whether a scan, a join, or another relational operation.
* **Plans –** a list of the ancestors for the current relational operator.
* **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.*

Now we going to parse the JSON, by recursively visiting operation ancestors. As we said, the ancestors are specified by the *PLANS* key, and an operator has no ancestor when the *PLANS* key is empty.   
  
For example, in Figure 6 these are the ancestors:

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 operation. This later will allow us to better understand the query.
   1. *Seq Scan\** on the titles which represent the titles after the filter, and it will instead of the *Seq Scan* in the ancestor hierarchy.
   2. *Seq Scan* on the titles which represent 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 *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 *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 represent the titles after the filter, and it will instead of the *Seq Scan* in the ancestor hierarchy.
   2. *Seq Scan* on the people which represent the titles before the filter, and will be the ancestor of the new *Seq Scan\* operator*.

After we parse the JSON we are going to save the information using tabular internal representation which is described in Figure 7.



**Figure 7**

As you can see, our table is a lot more concise, as we keep track of relevant information only. In addition, as we can drop some operations that don’t change the cardinality like *Gather and Hash* in the parsing step*.* The internal table representation includes the following information:

* **source/ target –** describe the ancestors’ hierarchy of a relational operator. The *source* column is an identifier of 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 information for a *Hash Join* is the join condition (*Hash Cond).*
* **actual\_rows –** is one of the metrics we want to measure that represent the subexpression cardinality.

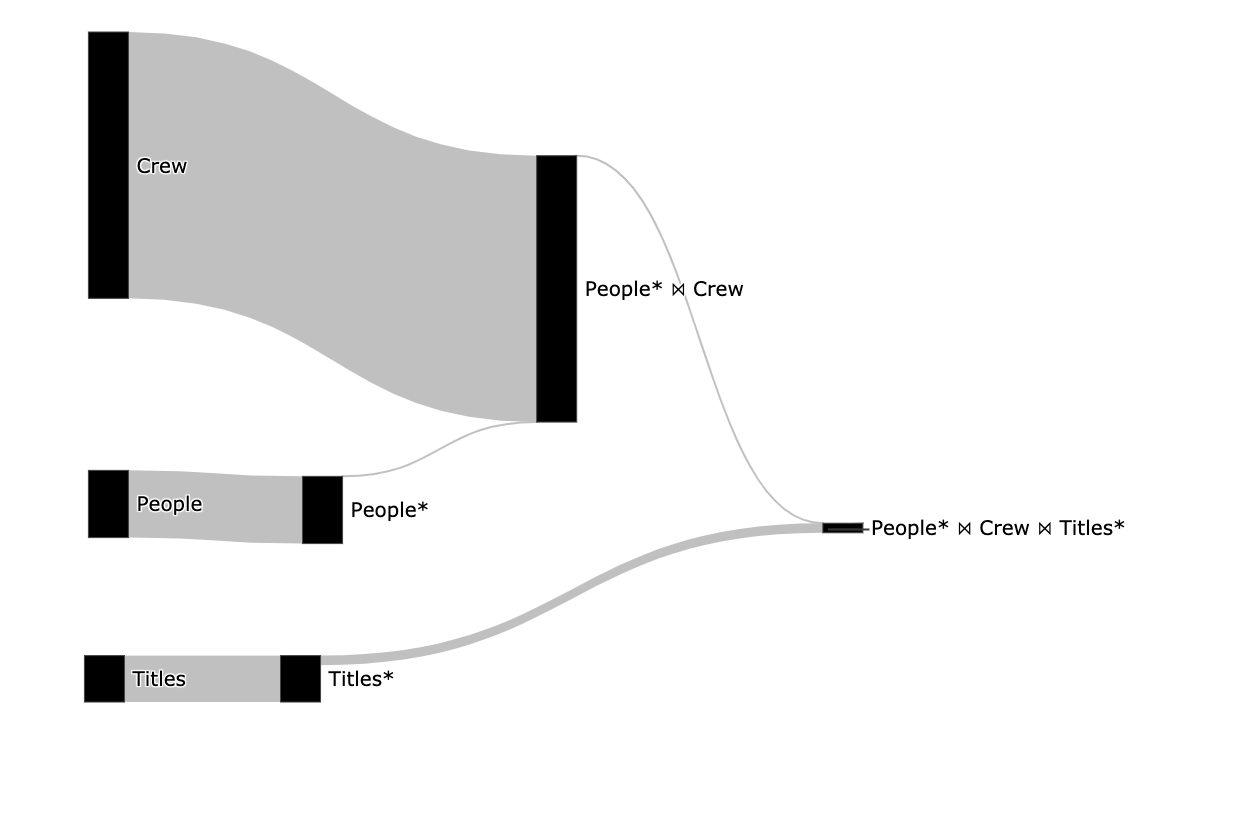
In our example, in Figure 7 we can see the following relations:

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.

But we still lack some can relevant information that will be entered in the enrichment process. The algorithm takes the table from Figure 7 and iterate the source and target as a BFS, and the iterations goes as follows.

1. We run on the row which represent the *People* subexpression, and since these don’t have ancestor there is no need for enrichment.
2. We run on the row which represent the *People\** subexpression.We calculate the *actual\_duration* by subtracting the *People total\_time* from *People\* total\_time.*
3. We run on the row which represent the *Crew* subexpression, and since these don’t have ancestor there is no need for enrichment.
4. We run on the row which represent the *People\* ⋈ Crew* subexpression. We calculate the *actual\_duration* by subtracting the maximum between *People\* total\_time* and *Crew total\_time* from the *People\* ⋈ Crew total\_time.* In addition, 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.*
5. We run on the row which represent the *Title* subexpression, and since these don’t have ancestor there is no need for enrichment.
6. We run on the row which represent the *Title\** subexpression.We calculate the *actual\_duration* by subtracting the *Title total\_time* from *Title\* total\_time.*
7. We run on the row which represent the *People\* ⋈ Crew ⋈ Title\* subexpression*. 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.* In addition, 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\*.*

Now that we got our internal table presentation, we are going to check whether there are special colors of our nodes and edge needed. Since there is no empty subexpression and no redundant subexpression, we will stick to the default coloring scheme.   
  
We are going to use *plot* function on our internal table representation to visualize our query as Sankey-diagram. The nodes will subexpression label. While edges, will represent a parent-child relationship, aka the operator that was applied to the relations and width proportional to the cardinality. Additional details regarding the operator are provided when hovering an edge.



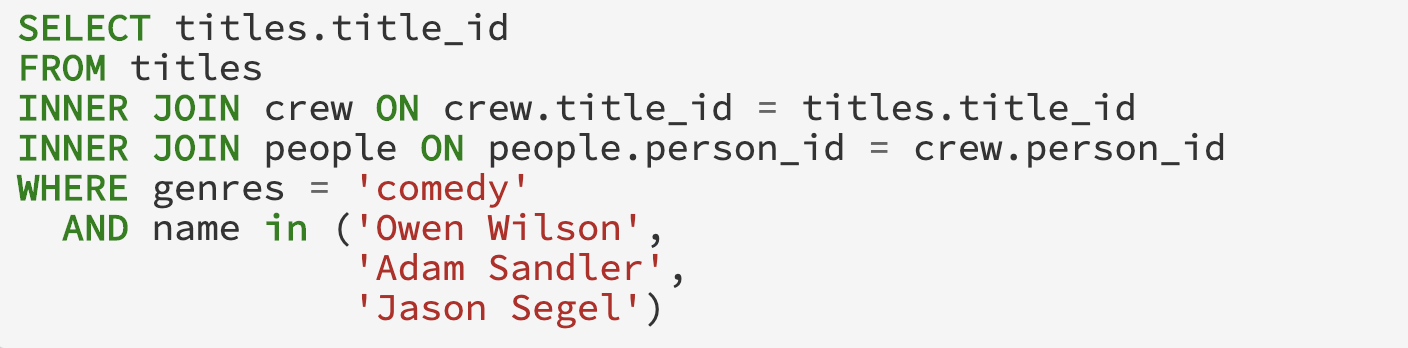
**Figure 8**

After we created the visualization, we can see which operator hierarchy and the cardinality of each operation, by how thick an edge is. Now we can understand why splitting the *Seq Scan* operator is valuable, I can understand how many rows the original relation had *titles* and *people* and the cardinality we got after the filter.  
  
Figure 8 allows us to understand the following insights:

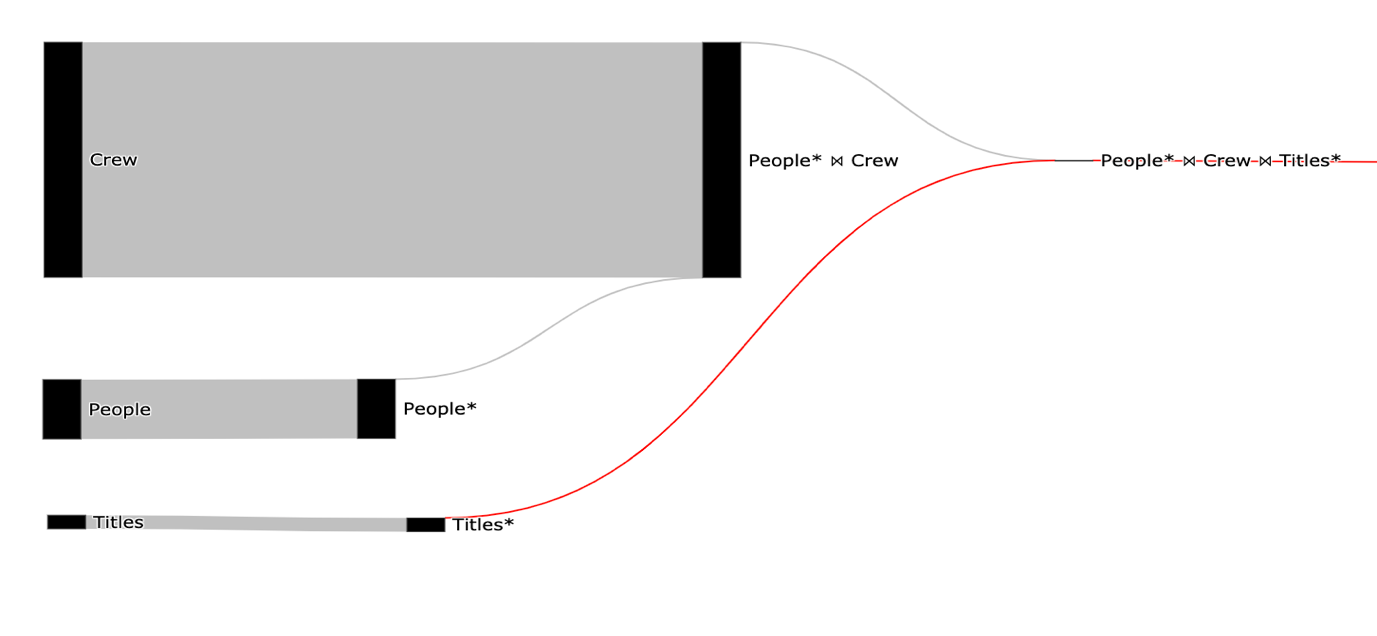
* The filter on the people relation is not redundant as the edge from it is thinner, and indices might be able to improve performance.
* The filter on the title relation is not redundant as the edge from it is thinner, and indices might be able to improve performance.
* No operation returns zero rows as no edge is red.
* Understand the size of the relations, the crew is by far the biggest relation.

Chapter 3: QueryFlow Use-cases

**3.1 Identifying missing records.**Problems 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 of the sub-expression of our query we can find missing records due to either WHERE, JOIN, UNION, or HAVING clauses.   
  
For example, in order to find out recommended movies for me, a possible query can be “find comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”. The SQL query and its corresponding Sankey for this question can be found in Figure 9 and 10.

****

­­  **Figure 9**

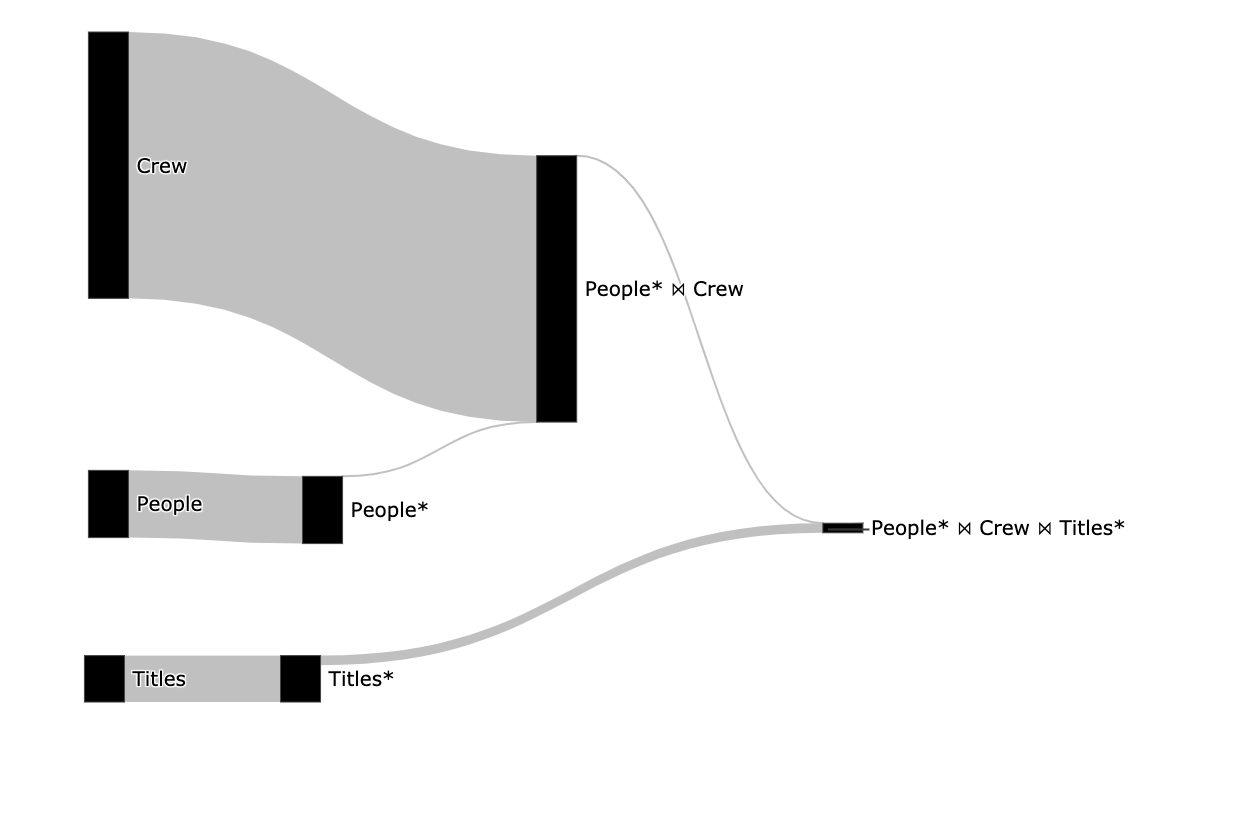
****

­­ **Figure 10**

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 rewrite the predicate to be Camel-Case instead of lower case and to support multiple genres in the same movie (as there is no lower case *comedy* value in genres). The fixed SQL query and its corresponding Sankey can be found in Figure 11 and 12.



­­ **Figure 11**



­­ **Figure 12**

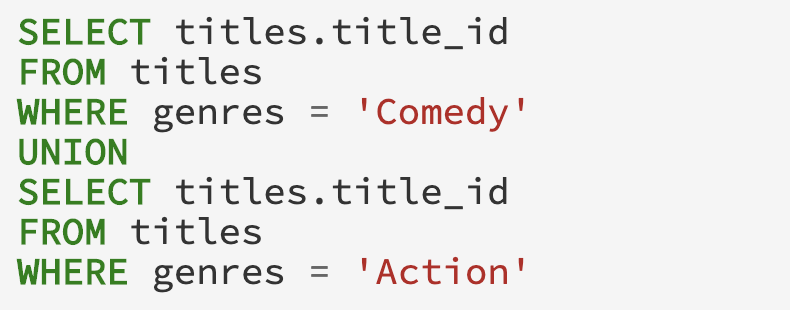
We can immediately see we fixed the problem and the insight that can be extracted by this figure are:

* The filter on the people/titles relations are not redundant as the edge from it is thinner.
* No operation returns zero rows as no edge is red.
* Understand the size of the relations.

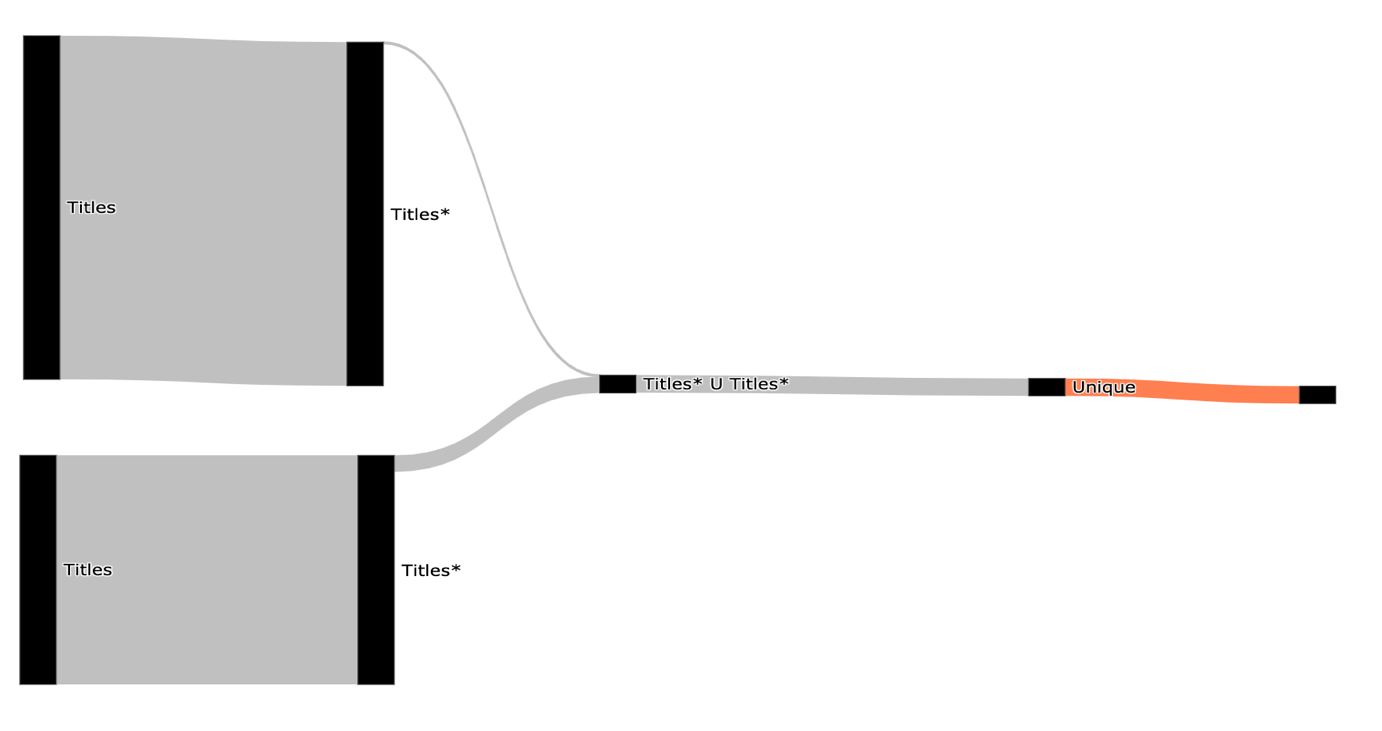
**3.2 Identifying Ineffective Operations**

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 of the sub-expression of our query we can find ineffective operations due to the *DISTINCT*, *UNION*, *WHERE,* and *HAVING* clauses.

For example, in order to find out recommended movies for me, a possible query can be “find all comedy movies and all action movies”. The SQL query and its corresponding Sankey for this question can be found in Figure 13 and 14.

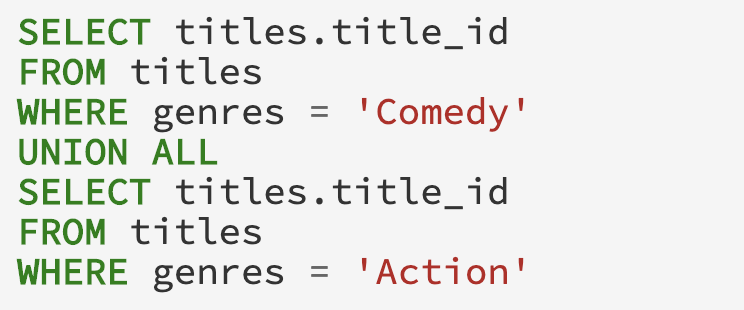


­­  **Figure 13**

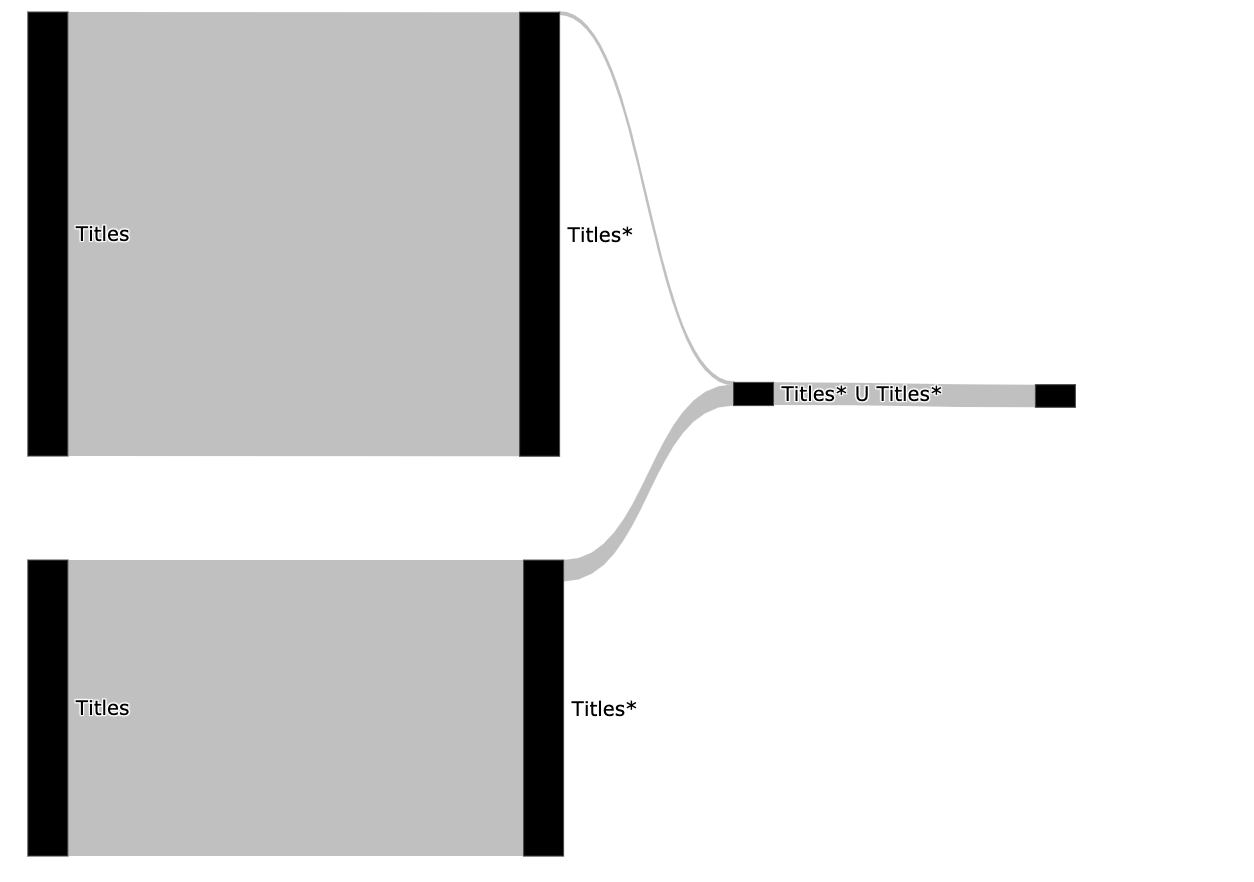


­­ **Figure 14**

Using the Sankey visualization, we can clearly see that the Unique operation 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.  
   
Now we can improve our query performance by modifying the *UNION* operation with a *UNION ALL* operation. The fixed SQL query and its corresponding Sankey can be found in Figure 15 and 16.



**Figure 15**



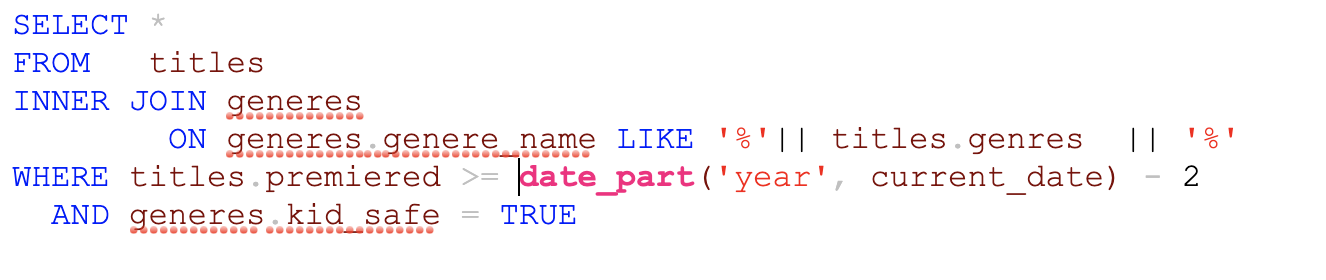
­­ **Figure 16**

We can immediately see we fixed our problem.

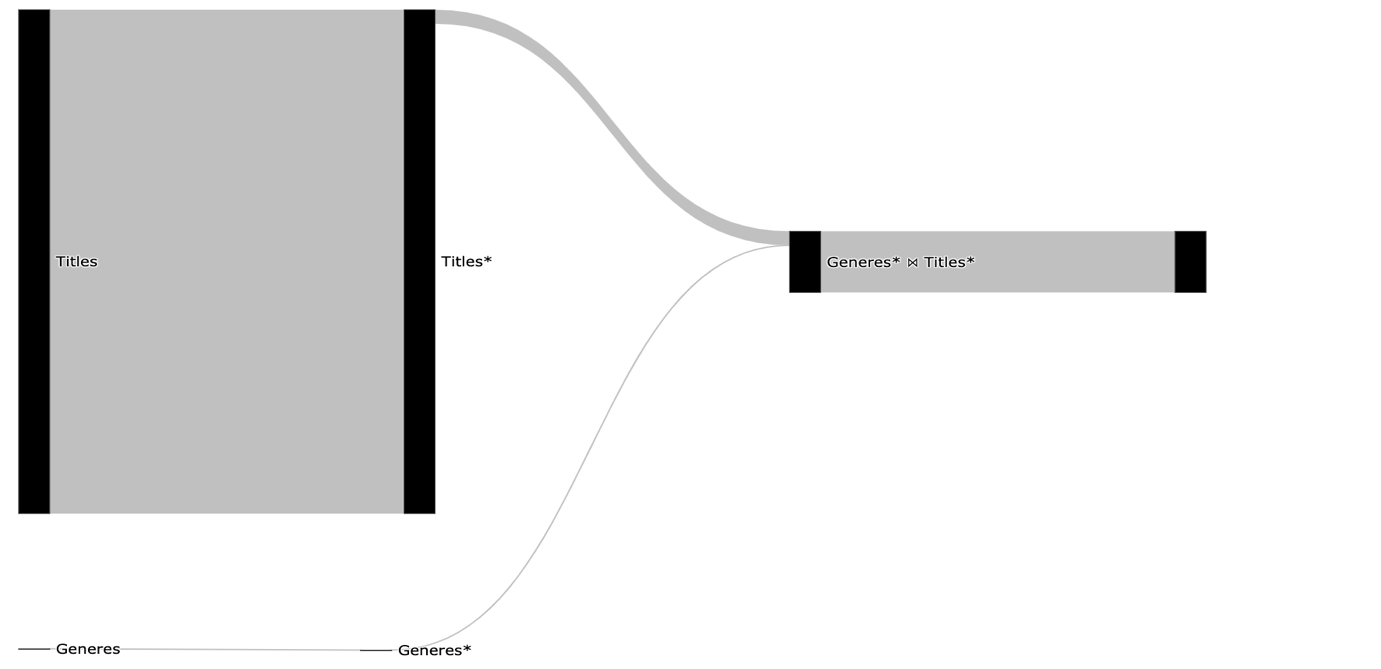
**3.3 Identifying Duplications**

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.

For example, in order to find out recommended movies for me and my family, a possible query can be “find all movies with genres that are safe for kids from the last two years”. The SQL query and its corresponding Sankey for this question can be found in Figure 17 and 18.



­­  **Figure 17**



­­ **Figure 18**

Using the Sankey visualization, we can clearly see that Join operation was exploding, these allow us to understand the JOIN condition is wrong and cause duplications.  
   
Now that we know we have a problem with the join clause, we can modify the query by adding a deduplication phase. The fixed SQL query and its corresponding Sankey can be found in Figure 19 and 20.

Text

Description automatically generated with medium confidence

**Figure 19**

Diagram

Description automatically generated

**Figure 20**

We can immediately see (Figure 20) the join introduces the duplications but after the subquery, we are removing those duplications.

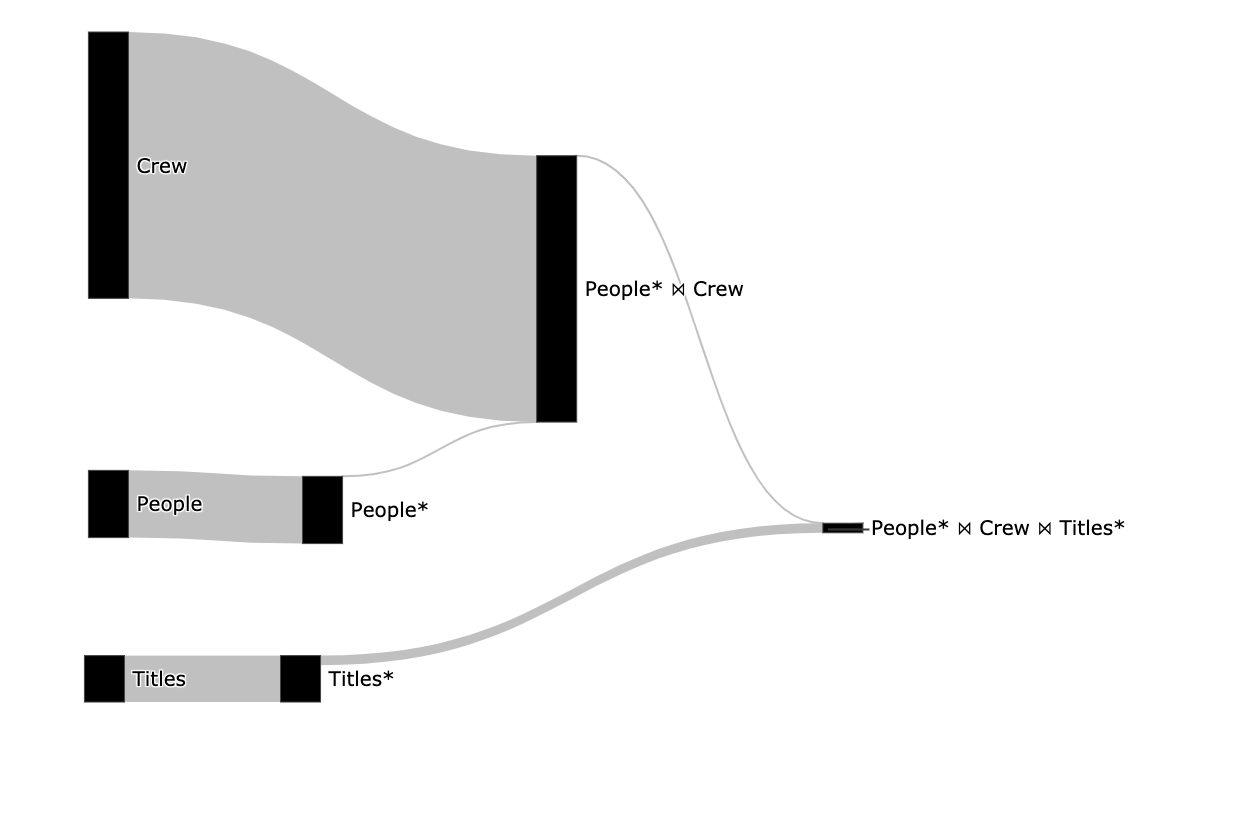
**3.4 Identifying Performance Bottlenecks in a Single Query**

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

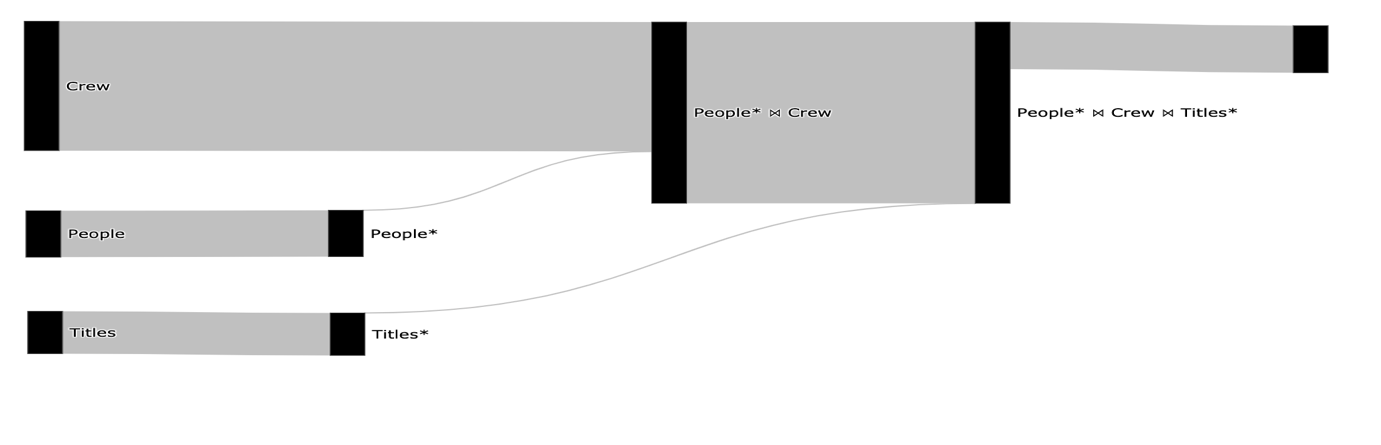
For example, in order to find out recommended movies for me, a possible query can be “find comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”. The SQL query can be found in figure 21, its corresponding cardinality Sankey can be found in figure 22 and the duration Sankey can be found in figure 23.



­­ **Figure 21**



­­ **Figure 22**



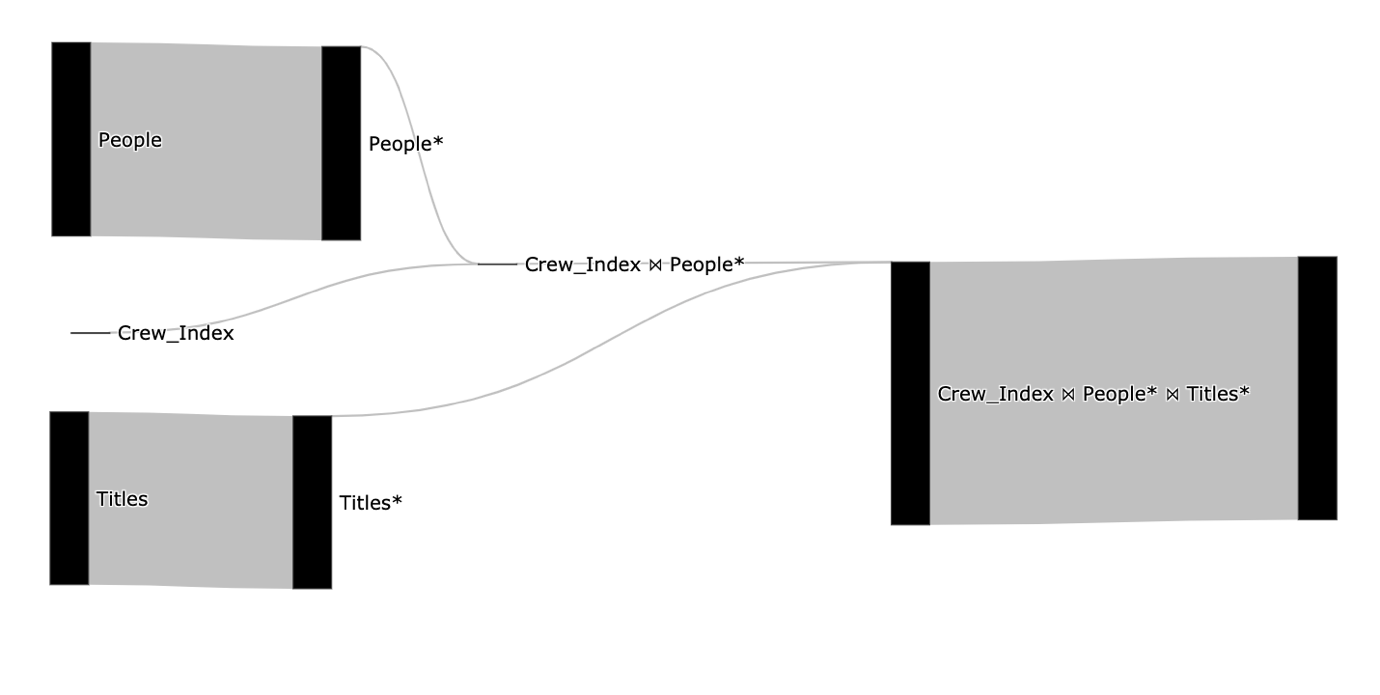
­­­­ **Figure 23**

The total query duration is 7 seconds, we can see that the longest operation (3.6 seconds) is the People\* ⋈ Crew (Figure 23), and we can see only a few rows are retrieved from the *People\* ⋈ Crew* due to the filter on the People relation (Figure 24). We can improve our query performance in multiple ways, one of them is to add an index on the crew relation using the *person\_id* column. The reason an index will improve our query execution time is the optimizer will be able to use *Hash Join* in a performant way.

The index creation query on *person* *id* and its corresponding Sankey of the duration can be found in Figure 24 and 25.



­­ **Figure 24**

****

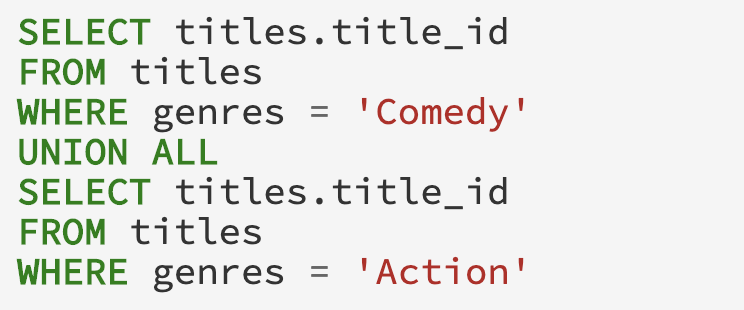
­­ **Figure 25**

The total query duration is 2 seconds, we can that it improved both the People\* ⋈ Crew and the scan on the *Crew* relation (Figure 25). We can continue to improve our query in the same manner if needed, for example we can improve the last Join by adding another index or by partitioning the *title* table.

**3.5 Identifying Performance Bottlenecks in Multiple Queries**   
To illustrate the gist of the MQO problem, we can consider these two queries:

* “Find all comedy movies and all action movies”, the SQL query can be found in Figure 26.
* “Find comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”, the SQL query can be found in Figure 27.

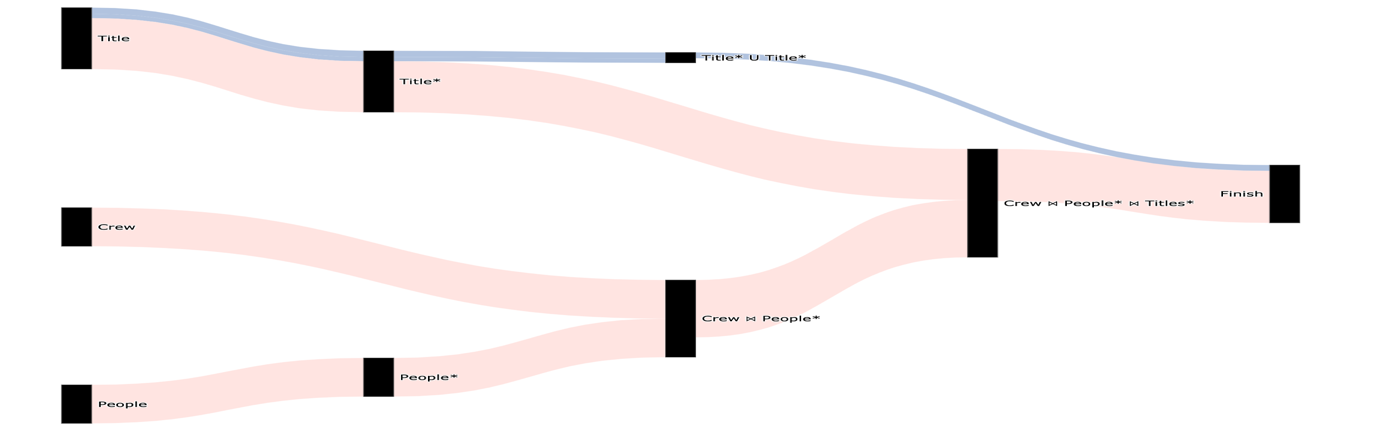
The cardinality and duration Sanky diagrams for both queries together can be found in Figure 28.



­­  **Figure 26**



­­  **Figure 27**

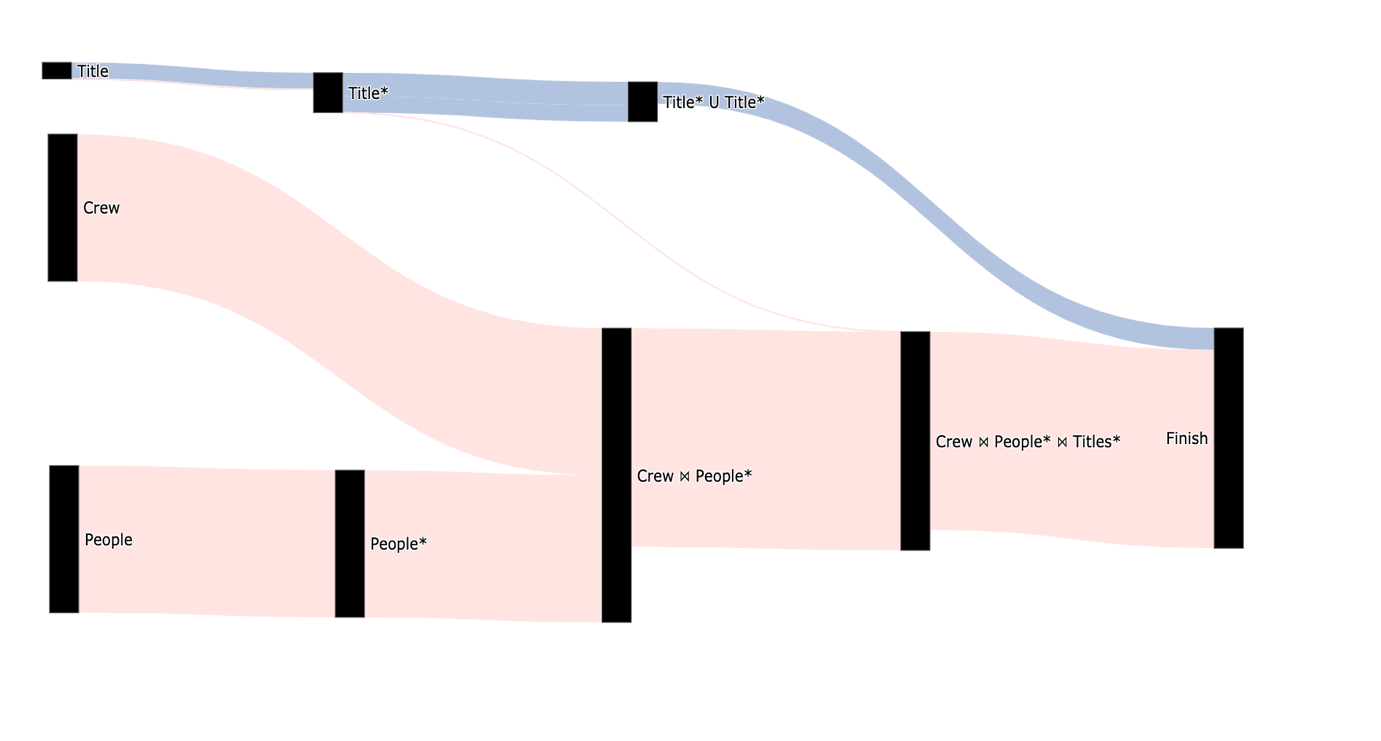


­­  **Figure 28**

The total query duration of both queries is 46.5 seconds, we can see that both queries are using the *Titles* and does filter on the genres (Figure 28).  
   
We can improve our query performance in multiple ways, one of them is to add a partial index on the *Titles* relation using the *title\_id* column on only Comedy and Action movies. The index creation query and its corresponding Sankey of the duration can be found in Figure 29 and 30.



­­ **Figure 29**

****

­­ **Figure 30**

The total query duration is 36 seconds, we can that it improved both queries as can be seen in Figure 30. We can continue to improve our query in the same manner if needed.

**3.6 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 the cardinality of the sub-expression of our query we can stale statistics and relations that need to be vacuum.   
  
For example, in order to find out recommended movies for me, a possible query can be “find comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”. The SQL query and its corresponding Sankey for this question can be found in Figure 31 and 32.   
Text

Description automatically generated

­­  **Figure 31**

Diagram

Description automatically generated

­­ **Figure 32**

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 that for the optimizer was way off for the *Crew* scan the (Figure 32), as the light gray edge is much thicker than the darker one.  
   
We can clean the dead tuple in the *Crew* relation using the vacuum command only on the Crew relation. The vacuum query and the corresponding Sankey after it can be found in Figure 33 and 34.

Graphical user interface, text, application, chat or text message

Description automatically generated

**Figure 33**

**Diagram

Description automatically generated**

­­ **Figure 34**

We can immediately see (Figure 34) the *Crew* scan is no longer skewed as used to be (Figure 32.

* 1. **When QueryFlow won’t help**

Like all tools, QueryFlow won’t be effective in all the cases like the famous quote says “There is no free lunch”. Since QueryFlow visualize query sub-expressions’ according to a measurable metric, there are edge cases where the metric visualization is not clear. For example, if one of the sub-expressions has extreme values, it will make the other parts of the query less visible.

One can try to mitigate it by moving to percentage representation, but in some extreme cases this will not be suffice.

Evaluation

The goal of this evaluation is to objectively quantify the impact and benefits that can be achieved using QueryFlow.

I’m going to use the TPC-H benchmark. It 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 by in academic, and the queries and the data have been chosen to have broad industry-wide relevance.

The tests were performed on a single processor TODO machine with TODO memory, running TODO. Optimization times are measured as CPU time (user+system). We used the TPC-H database at scale of TODO (i.e., TODO GB total size).

**Experiment**The workload for the first experiment consisted of TODO queries based on the TPC-H benchmark. Using QueryFlow we were able to optimize TODO out of TODO queries, the results are discussed below and plotted in Figure TODO.

TPC-H query Q-TODO has a BIG OPERATION-TODO. For this modified query, the actual duration of TODO secs., while we started with the actual duration of TODO, an improvement by almost a TODO.

TODO more queries While the benefits of using MQO show up on query workloads with common subexpressions, a relevant issue is the performance on workloads with rare or nonexistent overlaps. To study the overheads of Greedy in a case with no sharing, we took a batch containing TPCD queries Q3, Q5, Q7, Q9 and Q10, and renamed the relations to remove all overlaps between queries. Basic Volcano optimization took 650 msec, while the Greedy algorithm took 820 msec. Thu

To summarize, TODO.

Conclusions

In this paper, we introduced QueryFlow, a tool for query visualizations to ease identify flaws in user queries. Understanding those errors and bottlenecks can improve productivity and hints users towards causes they had not considered.   
  
To facilitate this task, we have described an approach that can automatically transform SQL queries into Sankey diagrams. This gives the users an intuitive understanding of the query characteristics by observing how the query is executed under the hood.

**References**

[1] 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.

[2] 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)

[3] 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)

[4] [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)

[5] [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)

[6] [W. Gatterbauer. Databases will visualize queries too. PVLDB, 4(12), 2011.](http://www.vldb.org/pvldb/vol4/p1498-gatterbauer.pdf)

[7] [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.

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

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

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

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