Abstract

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.

Intro

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 the query “track how the data flow”.   
  
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.

**Main contributions**

1. **Cardinality Visualization** - A new representation for queries as a Sankey diagram that allows us to understand the nature of a query and finding “cardinality issues” due to either 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. Another example, let’s say we are calculating the conversion rates of clicks and as we guard our query with a distinct operator, using this tool is trivial to find whether the distinct operator is needed or not.
2. **Execution Time Visualization** – A new representation for queries as a Sankey diagram that allows us to understand the nature of a query and finding bottlenecks and their reasons. 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 which allows understanding whether a better join strategy or more refreshed statistics may help.
3. **Optimizer statistics Visualization** – 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 which allows understanding whether it’s an issue with stale statistics or with the optimizer configuration.

**Background and Related work**

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

The first strategy is to infer query characteristics by analyzing the logical structure of the query. 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 [1] attempts to achieve exactly that like several others [2][3]. 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 of the actual data and requires the user to interpret the results to identify errors, and devise solutions.

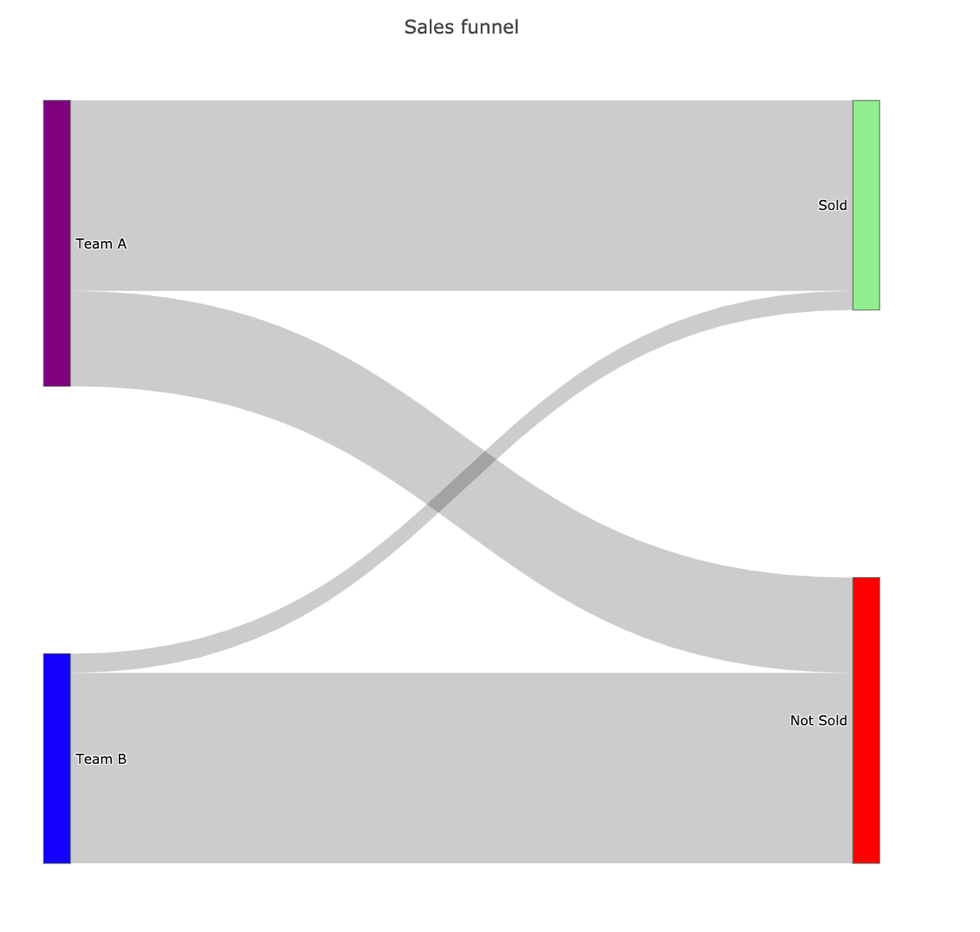
The second strategy is to infer query characteristics by analyzing the execution structure of the SQL query. Execution structure refers to the intermediate results and statistics at various stages of query execution. The techniques illustrate the structure of a query via a dynamic analysis of the query execution and focus only on execution time. For example, Picasso [4] attempts to understand the queries' bottlenecks as well as others [5]. 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, it still requires the user to interpret the results to identify errors, and devise solutions. Furthermore, since we actually executing the queries it might become resource-heavy for big data use-cases.

The third 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 [6]. But, it's will not provide a solution for issues in the actual query itself and thus is somewhat orthogonal to the previous two strategies.

The last strategy is to provide debugging capabilities for the database. The first technique is the why-not systems [7] 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 [8][9]. 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.  
  
Sankey diagrams [10] are a visualization technique that 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).

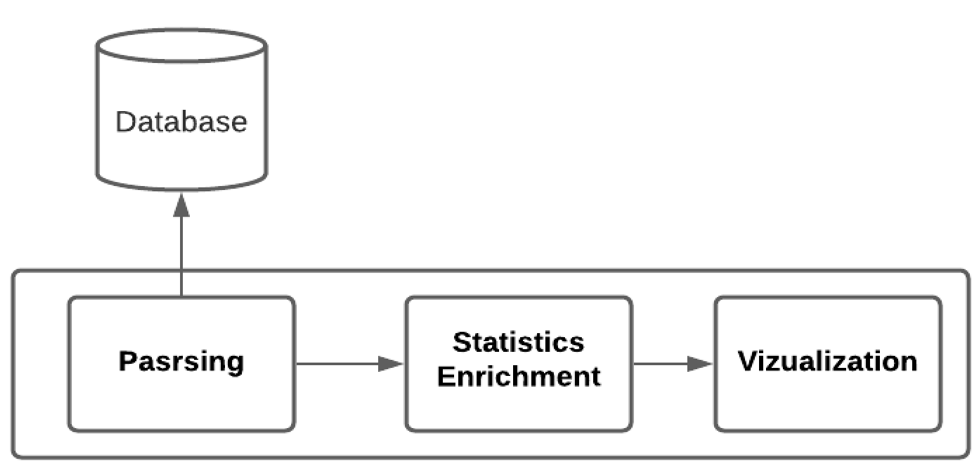
****

**Figure 2**

**The Design and Implementation of QueryFlow**

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 its 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 to get structured statistics
  2. **QueryFlow enrichment -** add additional statistics we can infer from the execution plan.
  3. **QueryFlow visualization**- visualize these characteristics and statistics using Sankey diagrams.

  
 **Figure 3**

**QueryFlow Parsing**

The parsing stage is implemented using python and we begin with executing an explain analyse query using SQLAlchemy, this will provide us with an execution plan with relevant and useful statistics of each sub-expression, for example, one statistic is the number of records the sub-expressions hold.   
  
We need to prepare the statistics for each sub-expression to be incorporated as a Sankey by creating a graph, for that 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.   
  
**QueryFlow Enrichment**  
Some very useful statistics are lacking from the execution plan in order to our comparisons, fortunately, these can be inferred.

* **Subexpression missing statistics-** Many statistics, like actual\_total\_time. actual\_total\_time and total\_cost are cumulative and include the ancestor sub-expressions and we want it in a sub-expression level.
* **Percentage statistics** - Representation of certain characteristics as a percent.
* **Redundant operations**- Whether an operation is effective or not, for example, a non-effective operation will be a distinct operation that filters nothing.

In order to enrich the statistics, I use the following algorithm:

1. Iterate each subexpression using BFS on the subexpression graph:
   1. For each subexpression collect its direct ancestors:
      1. For each missing subexpression statistic
         1. The missing statistic can be calculated by:  
             the current node cumulative statistic minus the maximum cumulative   
             statistic of the direct ancestors.
         2. The percentage representation can be calculated by:  
             the current node subexpression statistics divided by the current node   
             cumulative statistics and multiplied by hundred.
      2. If subexpression is of type “Unique “and the number of actual rows is equal   
          to the sum of the direct ancestors’ rows it is redundant.

**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 its 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:

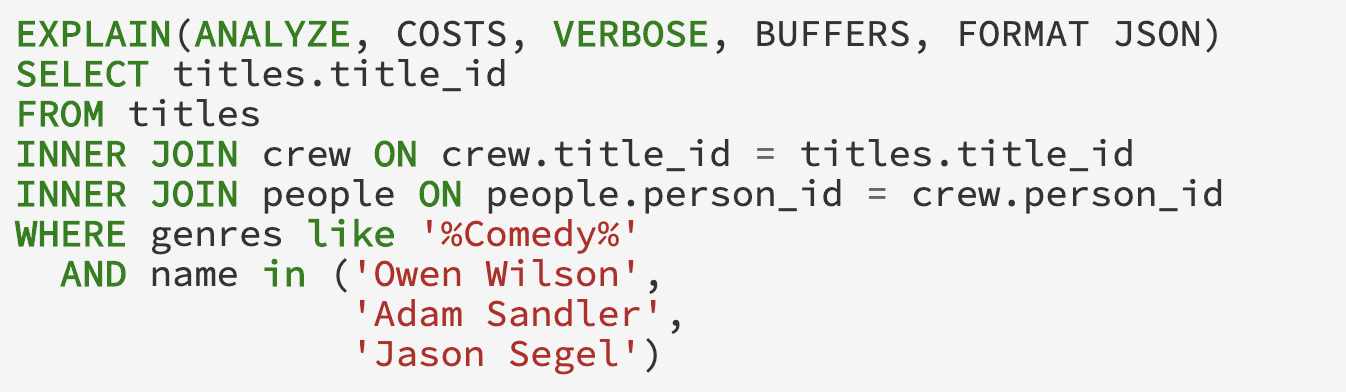
* When a Relation cardinality is zero.
* When an operation is redundant.

**QueryFlow Detailed Example**  
In order to illustration how QueryFlow works, I am going to use the IMDB dataset, and I am going to show you the cardinality visualization for a specific query. 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 (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)"* to the beginning of the query this will give us a new query which will return the actual execution plan, the new query 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. The actual execution plan JSON can be seen in Figure 6.  
  
 **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 which 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 has one ancestor, the “*Hash-Join”* between titles and crew.
2. The *“Hash-Join“* operation has two ancestors, the “*Seq Scan”* of titles and “*Hash”* operation. Since the “*Seq Scan”* has the filter key inside it, when we parse this operation we are going two split it into two
   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.
4. The *“Hash-Join“* operation has two ancestors, the “*Seq Scan”* of crew and “*Hash”* operation.
5. The “*Hash”* operation has one ancestor, the “*Seq Scan”* on people. Since it has the filter key inside it, when we parse this operation we are going two split it into two
   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 cardinality information using tabular internal representation which is described in Figure 7.



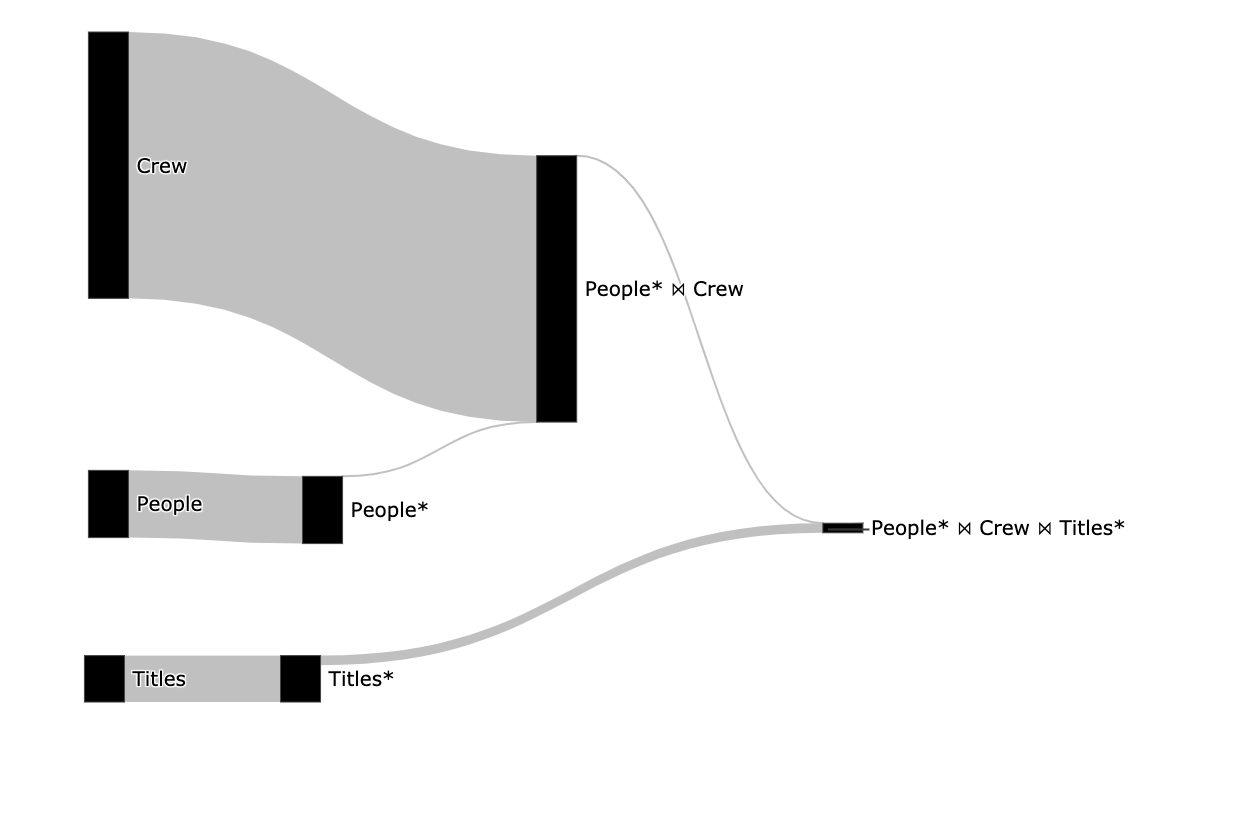
**Figure 7**  
   
As you can see, our table is a lot more concise, as we can drop some operations that don’t change the cardinality like *“Gather“ and “Hash“.* The internal table representation includes the following information:

* **source/ target –** describe the ancestors’ hierarchy of a relational operator.
* **operation\_type – is** the Node Type in the execution plan.
* **label –** logical representation of the operation type, this will allow us to group similar opertors like Hash Join and Merge Join on the same relation.
* **label\_metadata –** additional information of an operator.
* **actual\_rows –** the metric we want to measure.

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

1. The *“People“* node which represents the table scan has 3,446,261 rows and has an edge to *“People\*“*.
2. The *“People\*“* node which represents the table scan after the filter has 3 rows and has an edge to *“People\* ⋈ Crew“*.
3. The *“Crew“* node which represents the table scan has 13,651,901 rows and has an edge to *“People\* ⋈ Crew“*.
4. The *“People\* ⋈ Crew “* node has 565 rows and has an edge to *“People\* ⋈ Crew ⋈ Title“*.
5. The *“Titles“* node which represents the table scan has 489,076 rows and has an edge to *“Titles \*“*.
6. The *“Titles\*“* node which represents the table scan after the filter has 489,076 rows and has an edge to *“People\* ⋈ Crew ⋈ Title“*.
7. The *“People\* ⋈ Crew ⋈ Title“* node has 186 rows and it is a terminal node.

Now that we got our internal table presentation, we are going to visualize it, by using a Sankey Diagram as described in Figure 8.



**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.  
  
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.
* The filter on the title relation is 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.

QueryFlow Use-cases

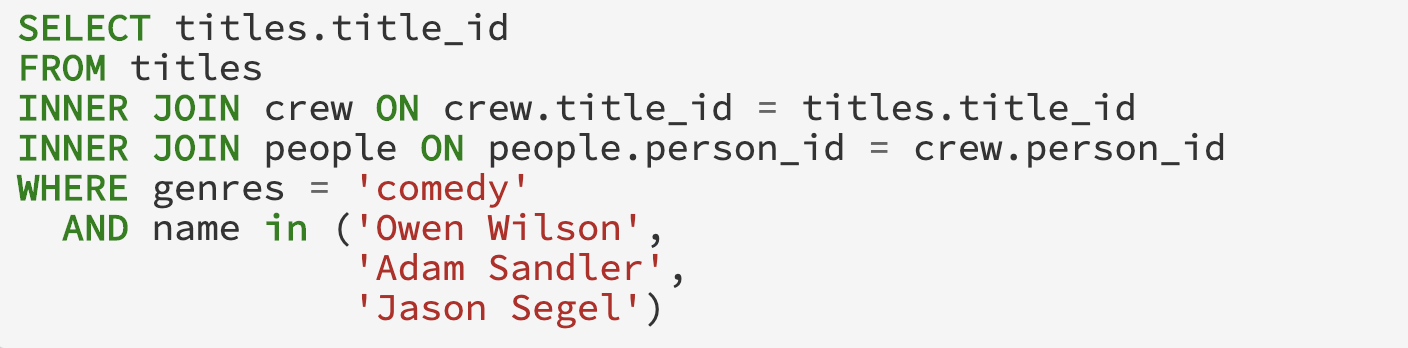
There are many types of mistakes that people tend to do, when composing their queries. In this section we are going to demonstrate the effectiveness of QueryFlow finding the following problems:

* Identifying missing records.
* Identifying Ineffective operations.
* Identifying duplications.
* Identifying performance bottlenecks in a single query.
* Identifying performance bottlenecks in multiple queries.
* Potential insight feed

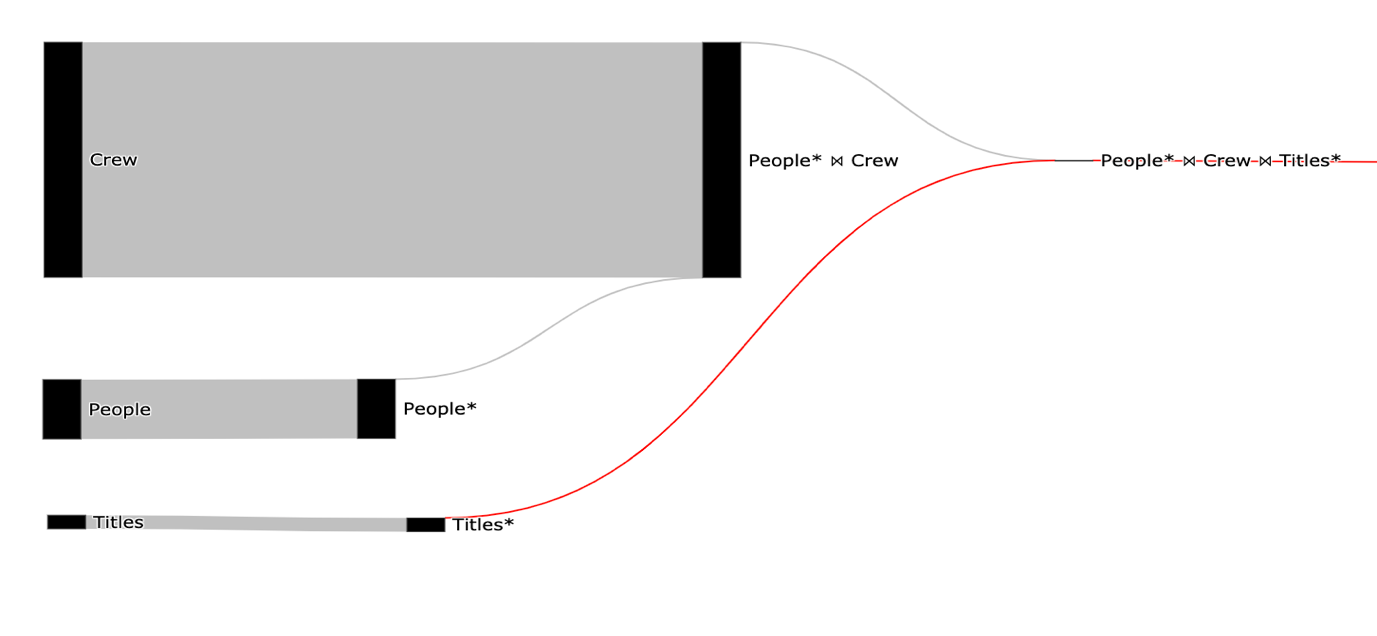
QueryFlow won’t be useful in all cases, like the famous quote says “There is no free lunch”.  
QueryFlow visualize query sub-expressions’ according to a measurable metric. In cases where one of the sub-expressions has extreme values, it will make the other parts of the query less visible. One can try to mitigate it by moving to percentage representation, but in some extreme cases this will not be suffice.

**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, if want to use the IMDB dataset to answer 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**

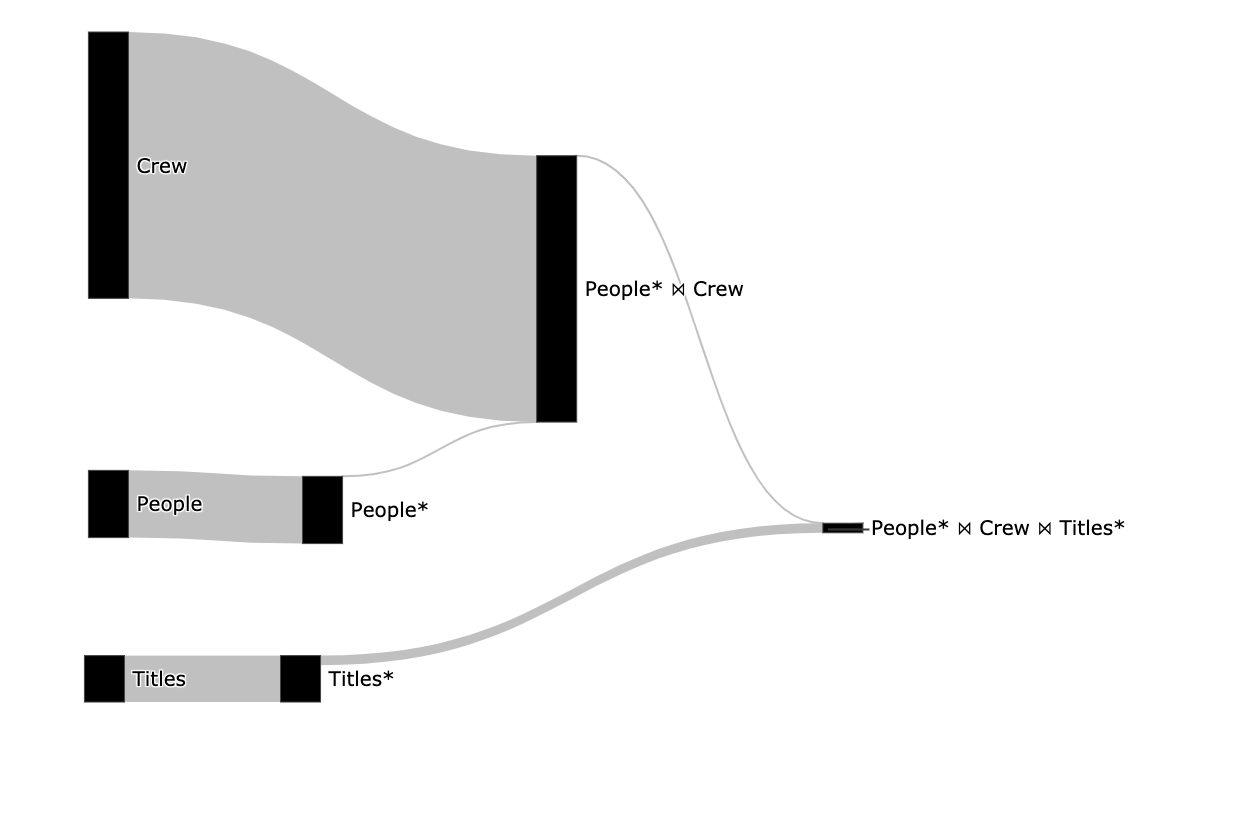
****

­­ **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. The fixed SQL query and its corresponding Sankey can be found in Figure 11 and 12.



­­ **Figure 11**



­­ **Figure 12**

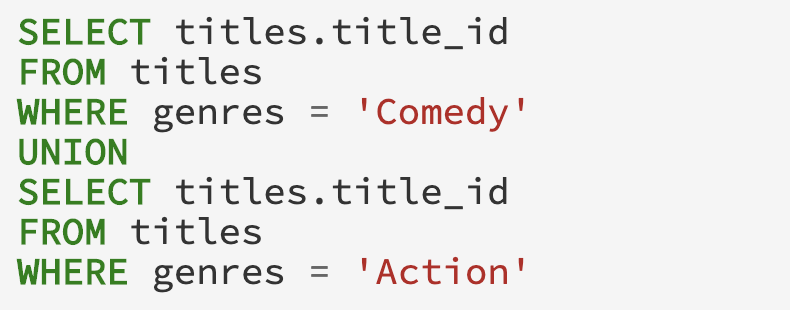
We can immediately see we fixed the problem and the insight that can be extracted by this figure are:

* 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.

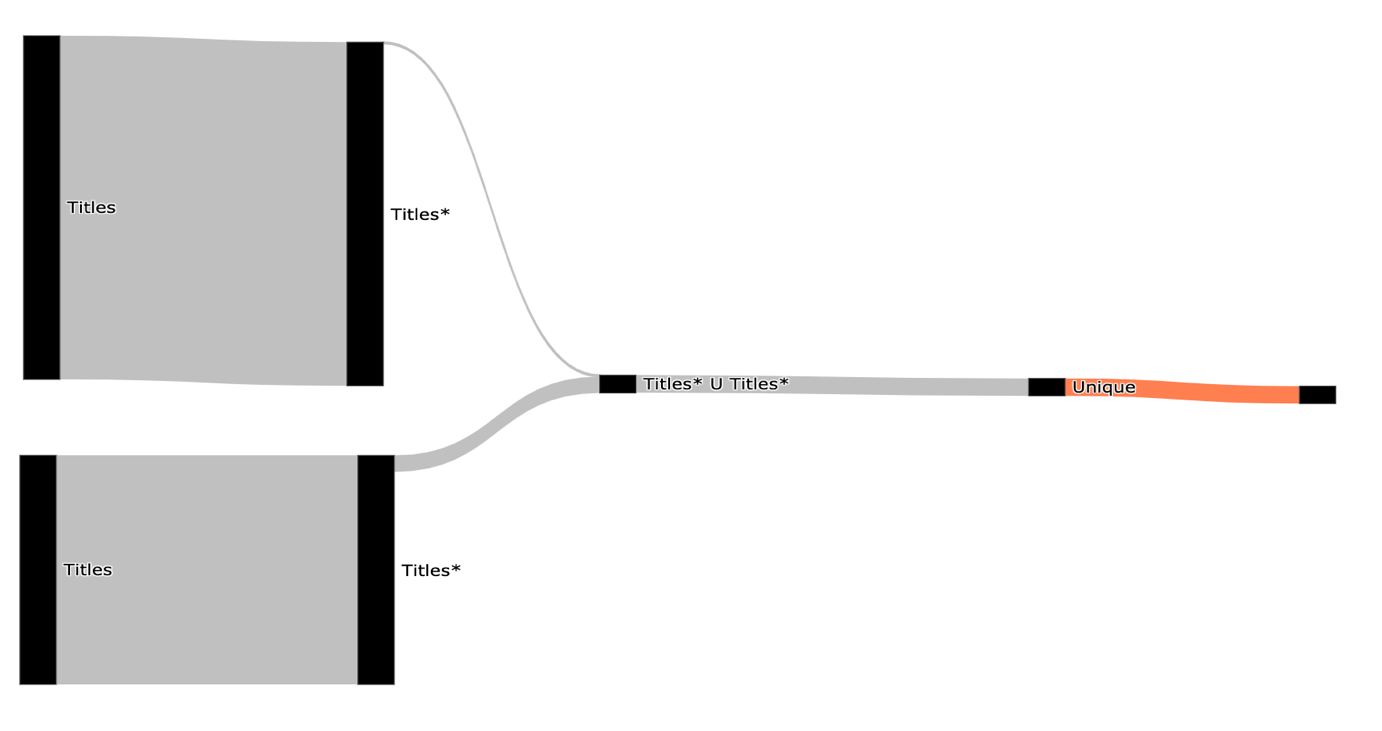
**Identifying Ineffective Operations**

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 clauses.

For example, if want to use the IMDB dataset to answer find out recommended movies for me, 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.

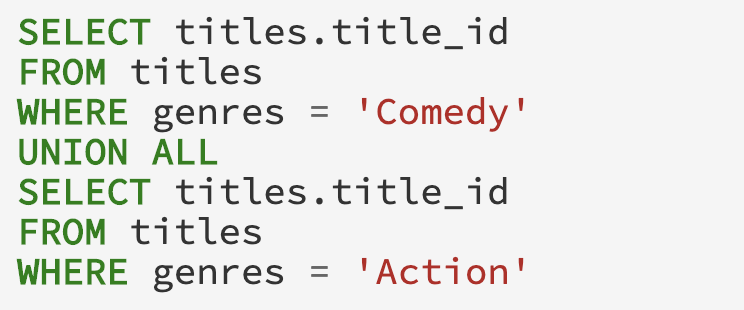


­­  **Figure 13**

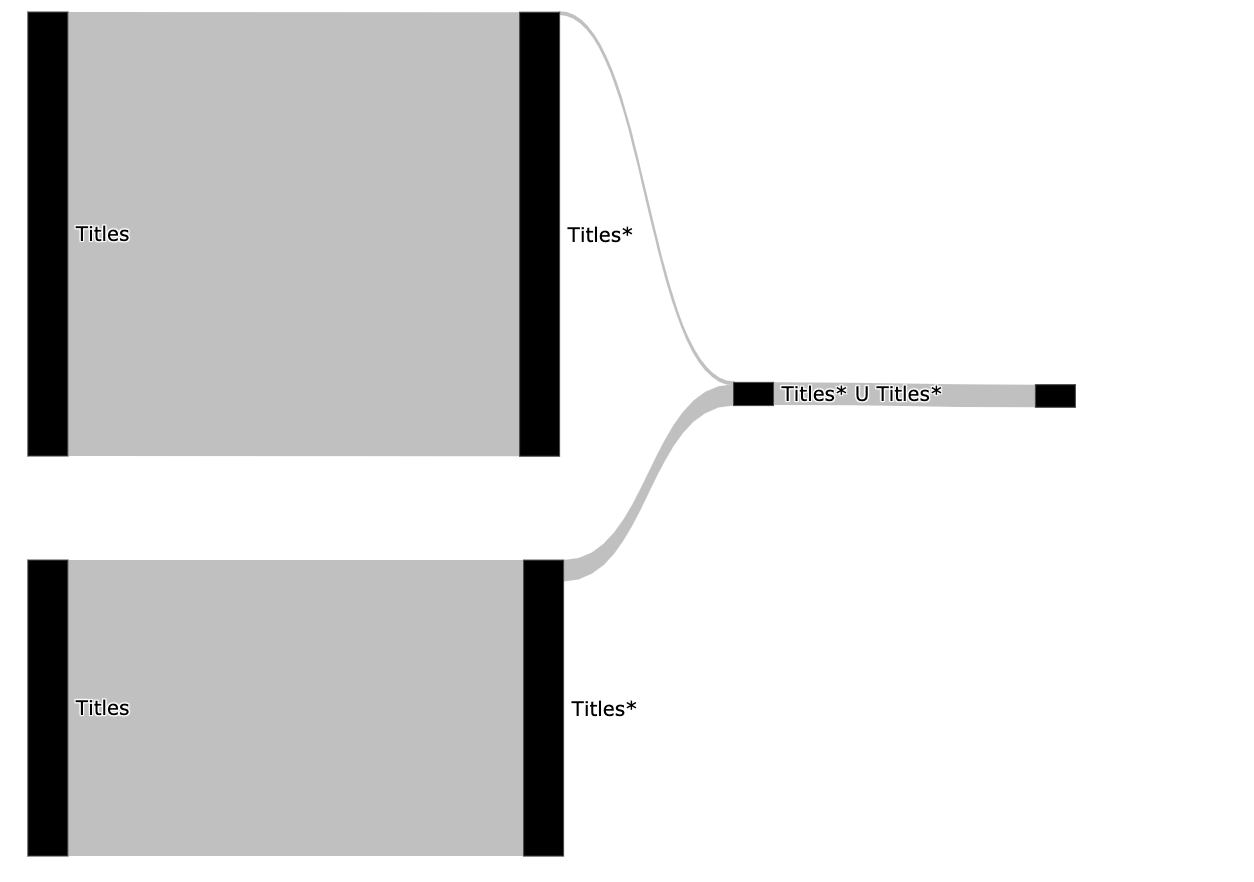


­­ **Figure 14**

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.  
   
Now we can improve our query performance by modifying the *“UNION”* operation with an *“UNION ALL”* operation. The fixed SQL query and its corresponding Sankey can be found in Figure 15 and 16.



**Figure 15**



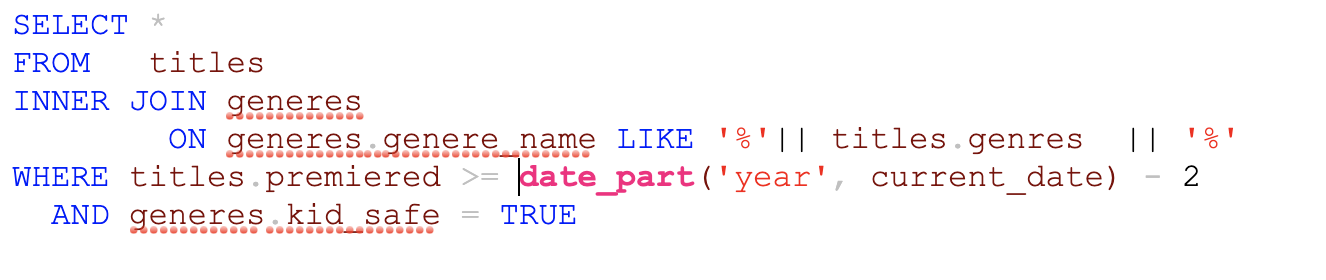
­­ **Figure 16**

We can immediately see we fixed our problem.

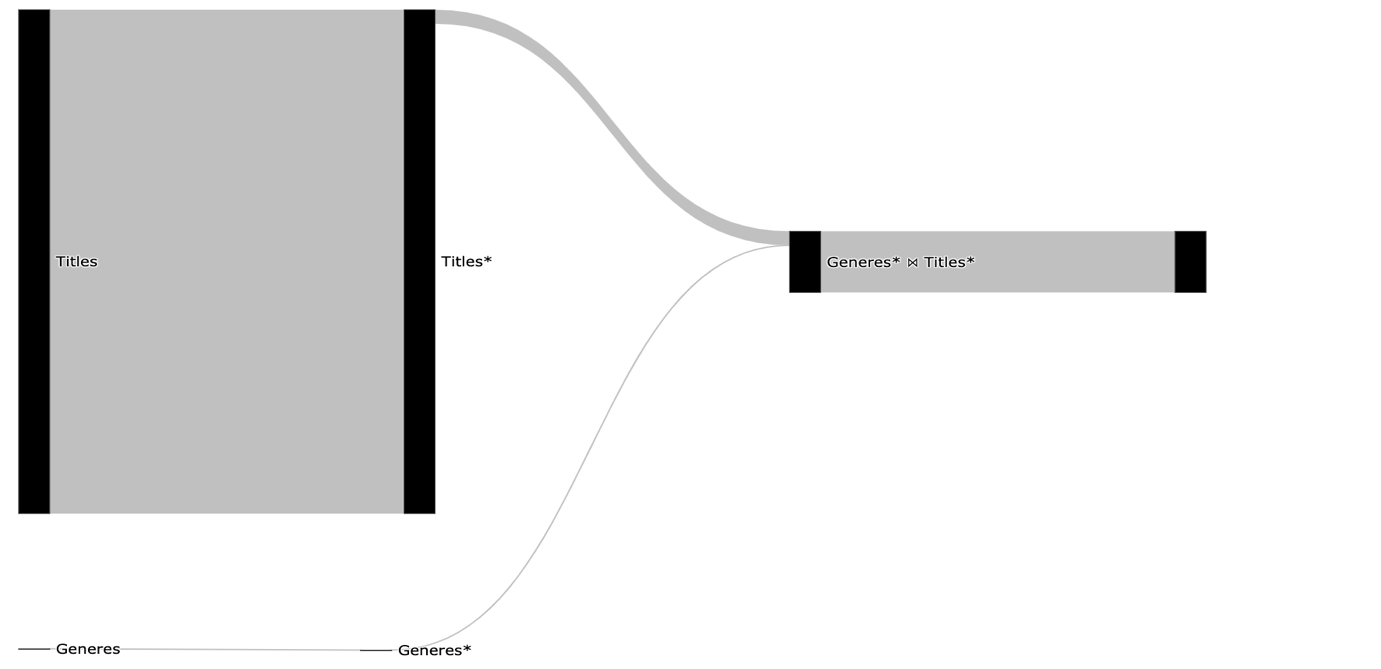
**Identifying Duplications**

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, if want to use the IMDB dataset to find out recommended movies for me and my family, a possible query can be “find all movies new movies with genres that are safe for kids”. The SQL query and its corresponding Sankey for this question can be found in Figure 17 and 18.

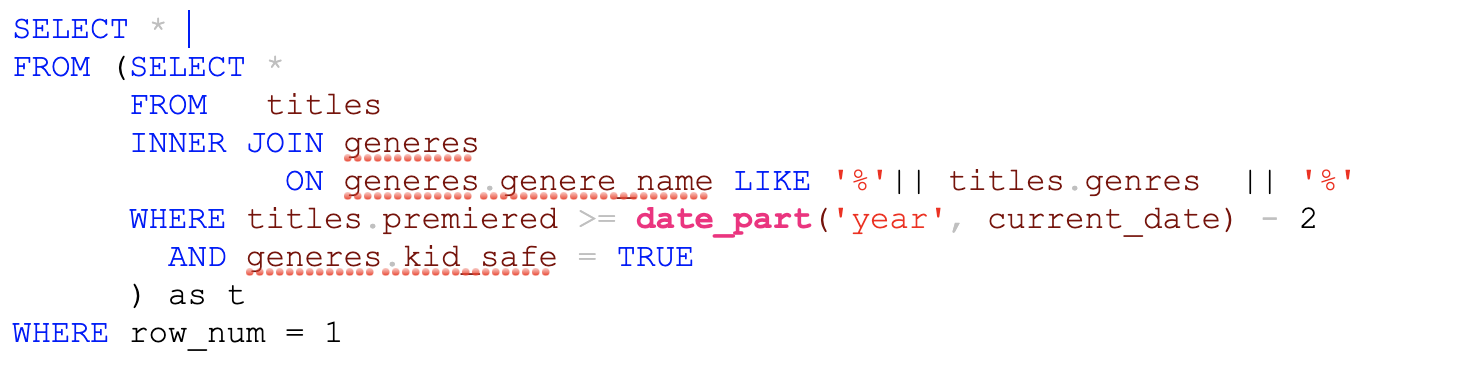


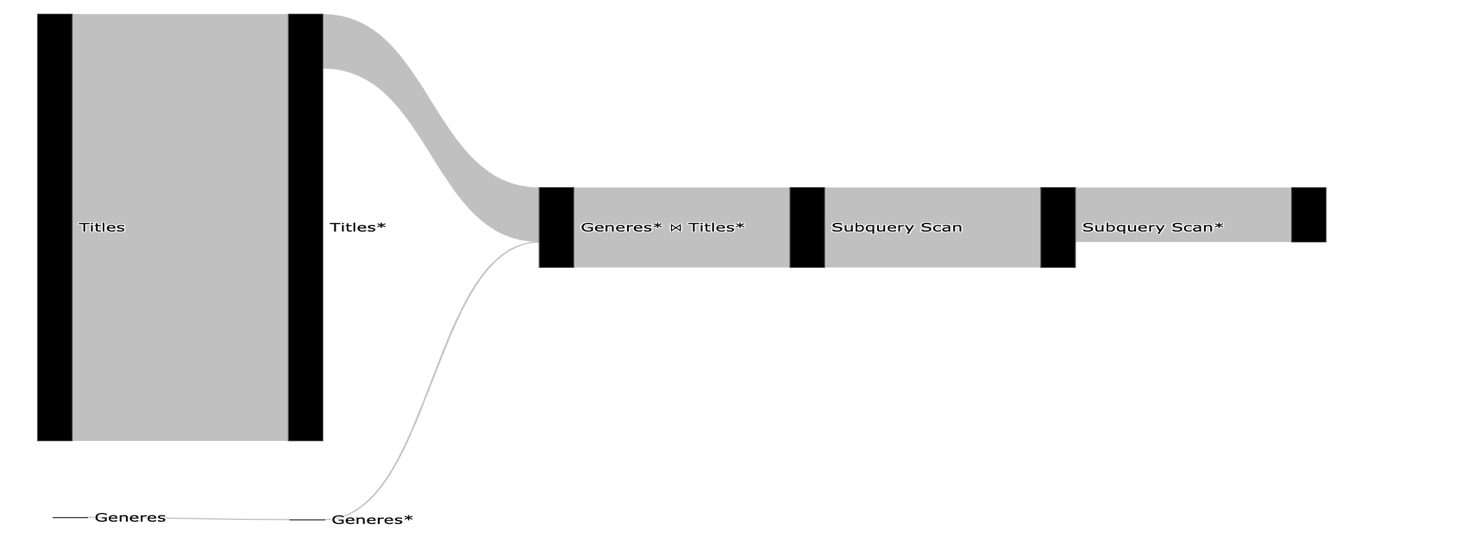
­­  **Figure 17**



­­ **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 15 and 16.



**Figure 19**

**Figure 20**

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

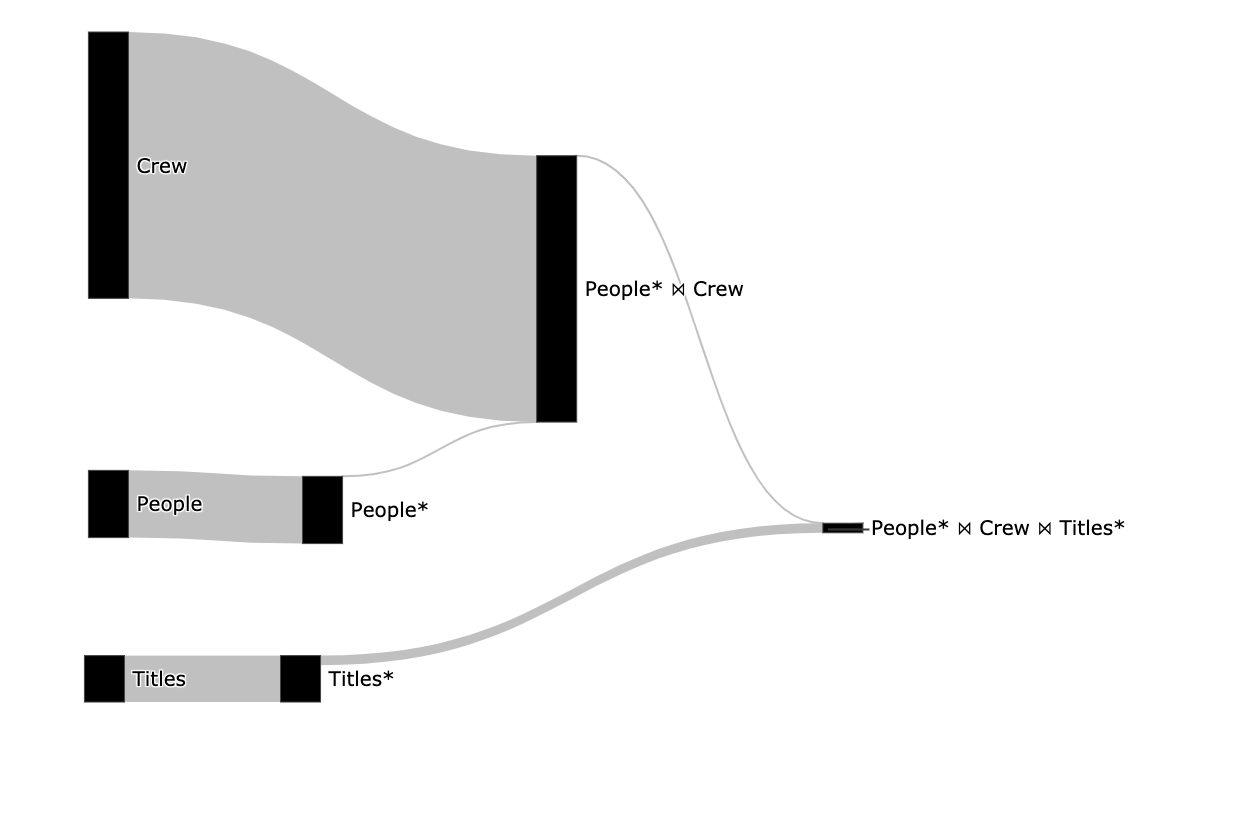
**Identifying Performance Bottlenecks in a Single Query**

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.

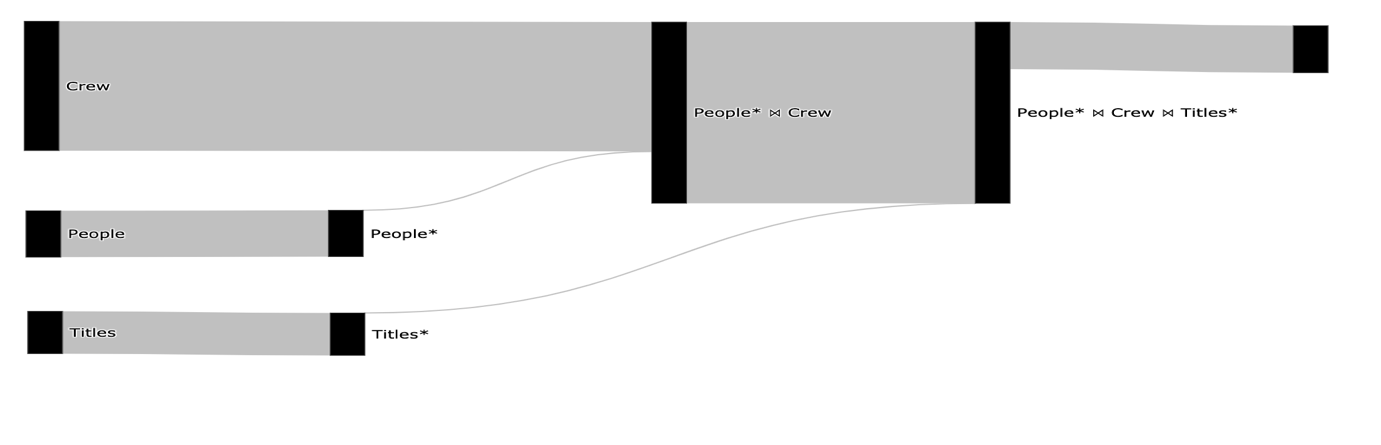
For example, if want to use the IMDB dataset to answer find out recommended movies for myself, a possible query can be “find comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”. The SQL query and its corresponding Sankies (cardinality and duration) for this question can be found in Figure 21, 22, and 23.



­­ **Figure 21**



­­ **Figure 22**

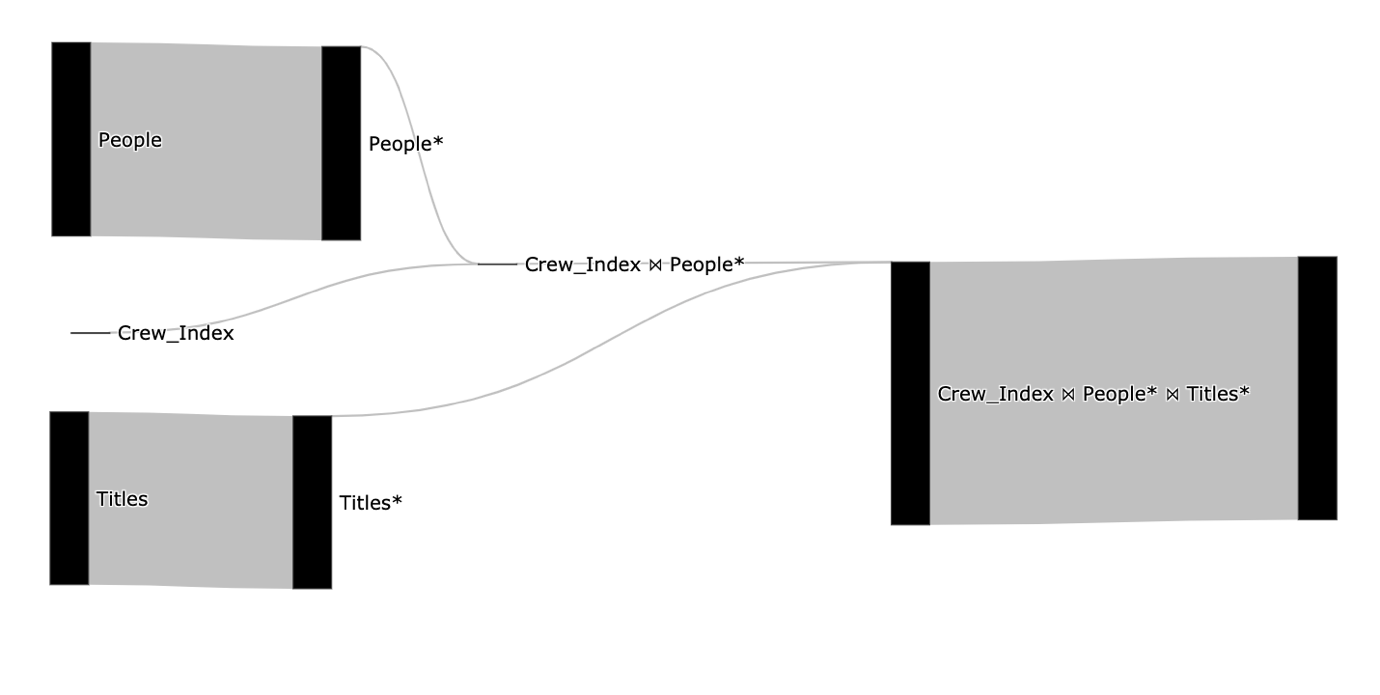


­­­­ **Figure 23**

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 index creation query and its corresponding Sankey of the duration can be found in Figure 24 and 25.



­­ **Figure 24**

****

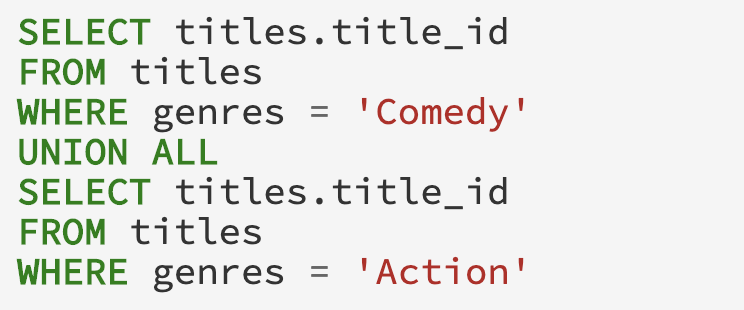
­­ **Figure 25**

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.

**Identifying Performance Bottlenecks in Multiple Queries**

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].   
  
We introduce a simple running example, that is rich enough to illustrate the gist of the MQO problem. Consider the following three concurrent queries:

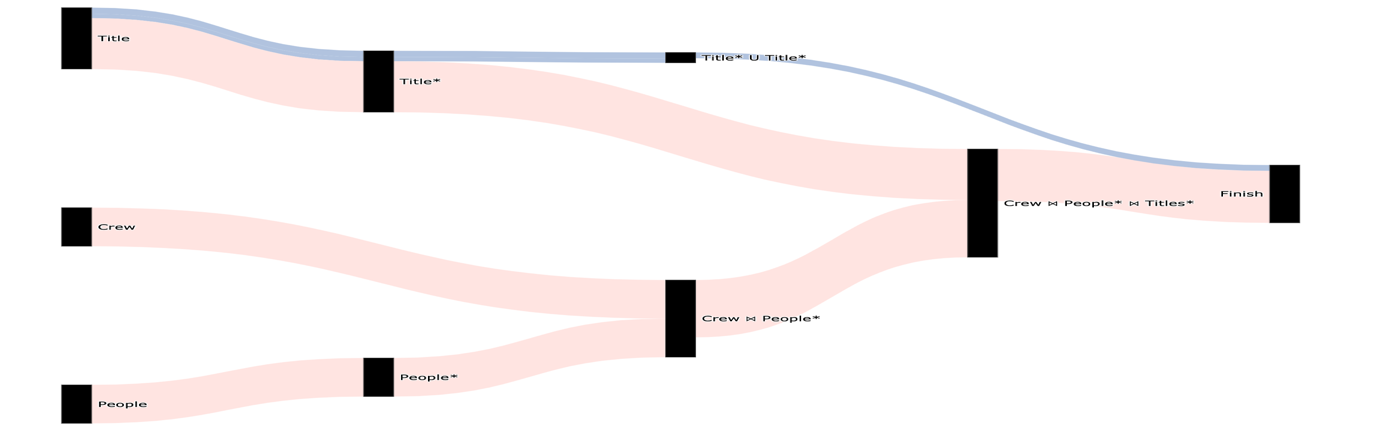
To illustrate the gist of the MQO problem, I can use the IMDB dataset for these two queries. The first is “Find all comedy movies and all action movies”, the SQL query can be found in Figure 26. The second is “Find comedy movies with Owen Wilson, Adam Sandler, or Jason Segel”, the SQL query can be found in Figure 27. The cardinality and duration sankies diagram for both queries together can be found in Figure 28.



­­  **Figure 26**



­­  **Figure 27**

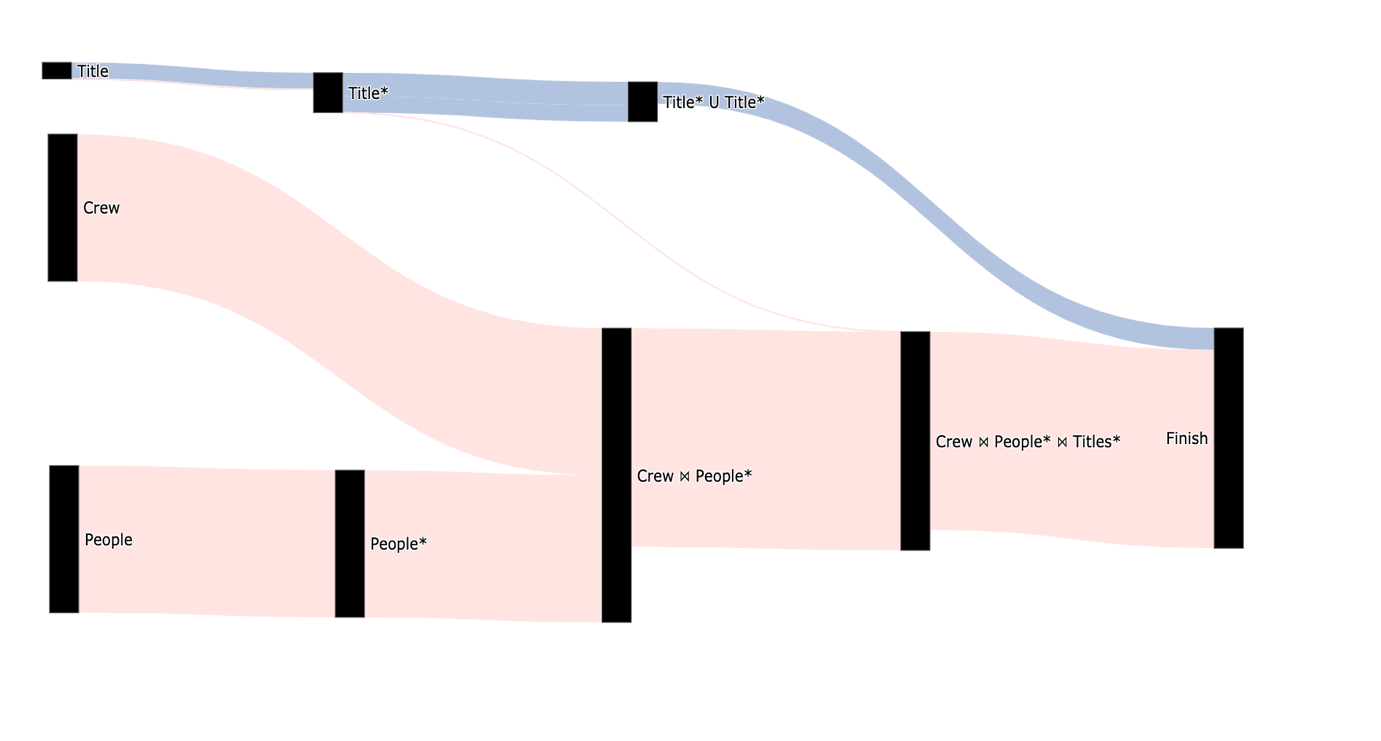


­­  **Figure 28**

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 an 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.

**Potential insights**

In order to make it even easier for users to understand where they should focus and what options exists, QueryFlow write a feed of potential interesting insights.

Some of the insights are:

- Sequential scans on large tables

- Filters that remove a lot of rows, partial indices  
  
- Planner estimates (planner missed by a factor of 100 or more)

- Excessive use of buffers  
  
- Corrupted indexes requiring a REINDEX

- work\_mem being set too low, preventing in-memory sorts and join

Evaluation

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

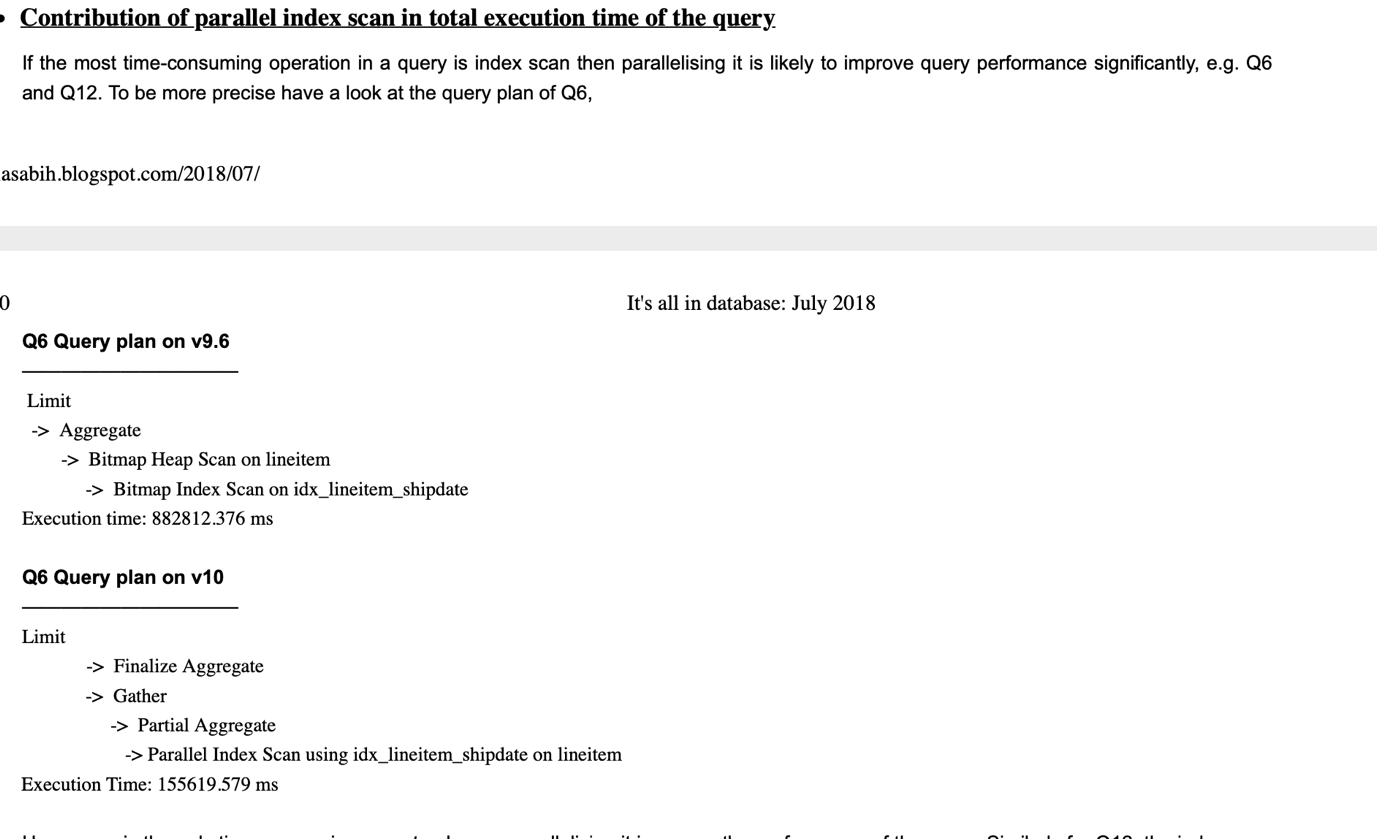
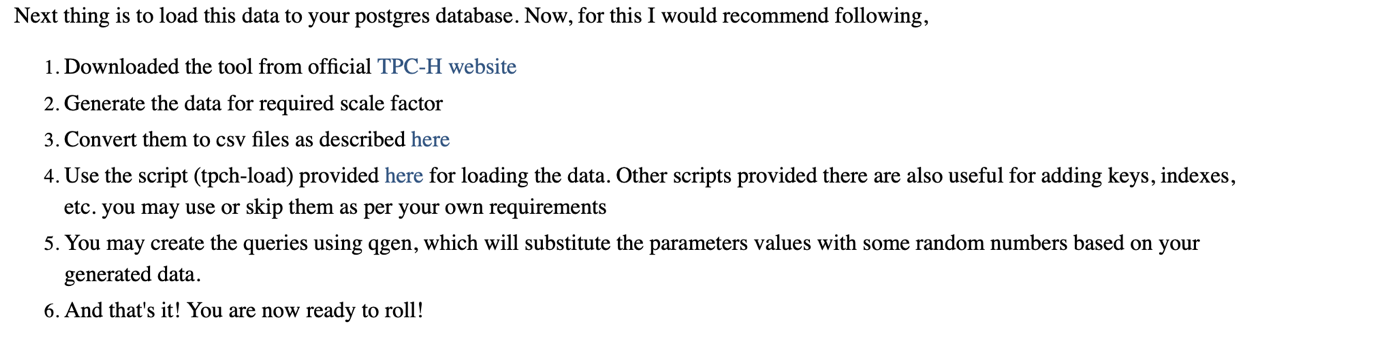
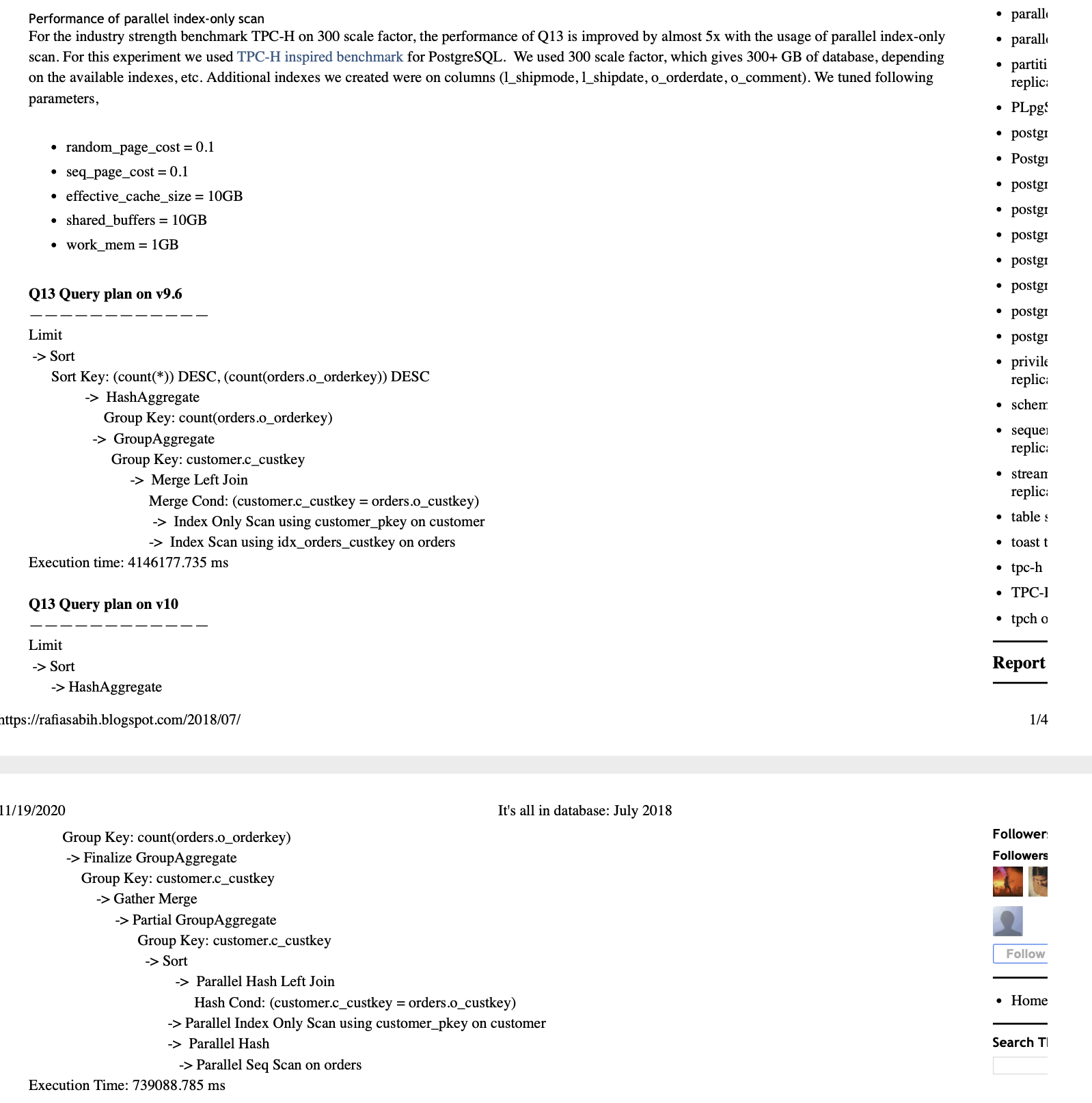
I’m going to use TPC-H benchmark. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications, that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The reason TPC-H was picked is because its 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 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] [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)

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

[3] [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.

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

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

[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] 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)

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

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

[10] 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.