QuiViz: Visualizing Physical Query Execution in a Relational Big Data Management System

Umar Javed, Thierry Moreau, Dominik Moritz, Adriana Szekeres
Dept. of Computer Science, University of Washington
Seattle, Washington, USA
{ujaved, moreau, domoritz, aaasz}@cs.washington.edu

ABSTRACT
We propose QuiViz, a visualization tool that helps database developers explore and understand query execution and data movement in a distributed database management system. QuiViz provides quick insight into common problems such as data skew or performance bottleneck by leveraging visualization techniques to present: (1) data flow between query operators and between workers, (2) query execution and operator dependencies, (3) cluster utilization, (4) network utilization.

In particular, QuiViz is built to inspect query execution in Myria [1], a distributed big data management system currently being developed in the UW CSE database group. Myria aims towards building a distributed database platform to provide *big data management and analytics as a service* primarily for scientific applications. QuiViz was designed to be easily ported to other DDBMS as well (e.g. Spark, Hadoop).

1. INTRODUCTION

Database query performance is often unpredictable, especially when the database is distributed. This requires database developers to spend significant amounts of time and effort isolating and debugging the root cause of performance problems. These may include skewed data partitioning or runtime issues, such as resource contention or adverse network conditions.

Our goal is to build a visualization tool designed to help a database developer understand the data flow and run-time performance during query execution in a Distributed Database Management System (DDBMS). Using our tool, the database developer could quickly discover run-time bottlenecks as well as problematic data skew issues, saving them significant guesswork and programming effort to identify the appropriate logs to analyze. Our visualization tool, QuiViz, uses profiling logs from *Myria* [1] as the prototype DDBMS. Myria is a distributed database platform that provides *big data management and analytics as a service* primarily for scientific applications. Myria takes queries in high-level query languages, such as SQL, translating them into physical query plans and finally executing them in parallel. A query plan is an ordered set of steps used to access data in database management systems. Myria breaks down each query plan into *fragments*. A fragment is composed of multiple sequential database operators running on one worker. Fragments map down to a unit of work that can be scheduled and parallelized across a cluster.

Currently, Myria does not offer a detailed visualization front-end designed to provide insight into job execution and network communication between distributed workers. QuiViz is designed to help the Myria developers write better optimized queries by addressing the following questions.

- *How does the data flow through the system between operators?* The overall data flow in a DDBMS is represented by a Physical Query Plan. It is important for the developer to understand the Physical Query Plan in order to optimize it. To aid in this understanding, we provide an interactive visualization of the Physical Query Plan in QuiViz.
- *How does the cluster visualization vary for each fragment and operator?* In *Myria*, a fragment is composed of multiple sequential database operators running on the same worker. A programmer will want to understand the cluster utilization at a given time as well as how much time is being spent executing an operator. QuiViz provides a Fragment Execution view to aid this understanding. The Fragment Execution view visualizes the utilization of the cluster by showing what operator in the fragment each worker is busy working on.
- *How is the input data partitioned?* Data skew is an important problem in a DDBMS, which can result in uneven communication between worker threads residing on different machines. QuiViz provides a view for Worker Communication that all-to-all communication between two fragments in the Physical Query Plan.

We show in the next sections how QuiViz addresses these questions by the means of effective data visualization.

2. RELATED WORK

Performance visualization tools for distributed systems are not a new idea.

Ambrose[11] is a platform developed by Twitter to visualize and real-time monitor MapReduce data workflows. Ambrose offers three different views to show associated jobs, job dependencies and progress, which are not suitable for our needs as the abstraction level of jobs is too high and does not capture single operators.

Google's Dapper[8] is a distributed systems tracing infrastructure that offers fine-grained tracing of calls in Google's distributed systems. They also proposed an interface for visualizing traces. Similarly, X-trace[5] was developed as a framework to trace which events cause what other events in a distributed environment. Recently, there has been work on visualizing event traces collected in X-trace[1]. Due to the importance of these kinds of debug facilities, Twitter closely modeled Zipkin[10] after Dapper and X-trace and released it as open source.

Dapper and X-trace focus on how data flows through a distributed system. In contrast, in QuiViz the visualization focuses on the operators and how the data flows through them. This orthogonal view is better suited for debugging performance bottlenecks in DDBMSs and also scales to a larger number of events. Furthermore, QuiViz offers different abstraction levels, which enables users to find problems faster and handle larger amounts of profiling data. Also, QuiViz is specifically designed to help developers understand query execution in distributed database system as opposed to general traces in distributed systems.

Tools to visualize query plans, for example those used to improve the performance for the SDSS Sky survey[9], focus on optimizing queries and not query execution and have no visualization of data flow, which is necessary to optimize physical query execution.

3. APPROACH

The first step in designing QuiViz was to identify the possible causes that affect the performance of a query execution. After we identified several such causes, we explored visualization techniques we found adequate in exposing those performance bottlenecks.

We found that the performance of a query might be affected by the following factors, which greatly influenced the design of our visualization tool:

- *Wrong/unoptimized physical query plan.* We manually analyzed several physical query plans and discovered that some queries were poorly optimized. This led us to design a view that allows interactive exploration of the physical query plan (mapped to a graph of query operators).
- *Stragglers, tail latency, expensive query operators, poor scheduling.* We target *distributed* database management systems. Therefore, as the query gets executed in a distributed environment, the query coordinator has to wait until all the workers finish executing their assigned tasks. To give insight into how the tasks are distributed and executed on each worker, we provide a view that exposes utilization details at the cluster and worker levels.
- *Poor data partitioning, data skew.* Network traffic can cause serious delay when executing a distributed query. Due to poor data partitioning, workers might need to send large amounts of data to one another. We designed a view that allows the developer to analyze the data traffic generated while executing the query.

[1]`https://github.com/brownsys/X-Trace/tree/master/src/webui/html/interactive`

3.1 System Overview

QuiViz's architecture (Figure 1) is composed of: (1) a back-end used as a plug-in interface to the targeted distributed database system, and (2) a platform-independent front-end that produces performance visualizations what we describe in Section 3.3. The front-end and back-end of QuiViz are highly decoupled so that the visualization can be used not only with Myria but also with other systems such as Spark or Hadoop. In Section 3.2 we illustrate how we implemented log collections and aggregation in Myria.

3.2 Back-end

The role of QuiViz's back-end is to produce event logs that include: when operators are called, when the call returns and when data is sent between workers. These logs are then used to create the visualizations in QuiViz's web UI as described in Section 3.3.

We explored two approaches for collecting event logs. The simplest approach was to write the logs on files directly to disk. The files would then get hauled back to one master node using remote copy where they would then get parsed, manipulated to the desired format.

We switched to a more scalable data collection approach described in Figure 1 where logs get written directly into the local worker's database as tuples in a relation. When the front-end client requests data, the master executes a query that gathers the log data from the dirstributed workers, and streams the results directly to the requesting client. We found this data collection approach to be more scalable, reliable and faster than the simpler file-based logging approach. More particularly, using a database to manage performance logs allowed us to write queries to aggregate, filter and transform the collected data.

For example, to produce the visualization described in Section 3.3.2, we had to query how many workers are working on a certain fragment at a given time. To do this, we used a query to select events in the root operators (i.e. parentless operator) of each fragment and used a custom map function to carry state. The state is an integer that is incremented when the operator is called on a worker and decremented when the operator returns.

3.3 Front-end

The Myria web front-end server is written in Python and runs on Google App Engine[2]. QuiViz's user-interface (UI) is embedded into Myria's web front-end. We build QuiViz's UI using D3[3] to deliver an interactive visualization experience to the user. D3 is a JavaScript framework for data visualization on the web.

QuiViz's web UI is divided into two components: (1) the browser panel which provides a visualization of the Physical Query Plan, and (2) the performance panel which provides some insight into an element selected in the browser view. The browser panel contains a view of the Physical Query

[2]`https://developers.google.com/appengine/`

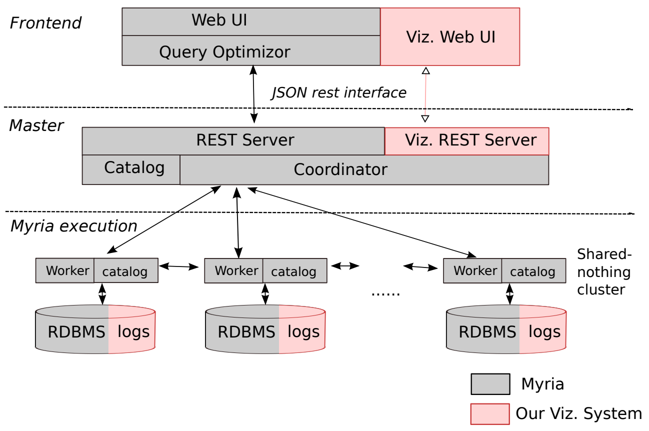
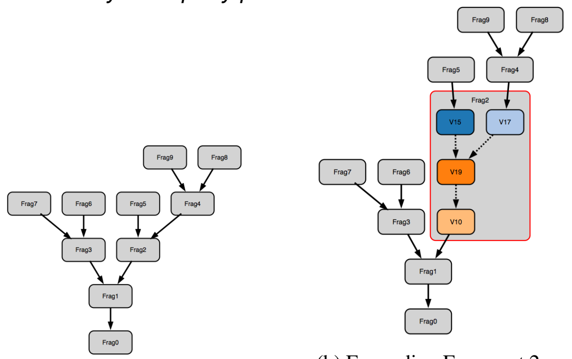


Figure 1: Overview of the log collection and transformation architecture. Raw event logs are collected on each worker. The data is stored in relations. To download the data, a query has to be executed.

Plan, rendered as a graph. The user can navigate the Physical Query Plan by expanding query fragments into the operators that compose it. The user can chose to select a fragment of interest which will render a Fragment Execution visualization of the selected fragment in the performance panel. Alternatively, the user can select an fragment-to-fragment edge in the Physical Query Plan thus rendering a Worker Communication visualization of the selected edge in the performance panel. If no elements are selected in the Physical Query Plan view, an Fragment Overview visualization is rendered in the performance panel, displaying the aggregate worker utilization over time for each fragment.

The following subsections describe each one of the four views offered by QuiViz's web UI.

3.3.1 Physical query plan view



(a) Large query plan collapsed.

(b) Expanding Fragment 2 reveals its operators.

Figure 2: The Physical Query Plan view is used by QuiViz to help the user browse performance visualizations.

The Physical Query Plan view as pictured in Figure 2 allows the user to selectively browse different performance visualizations of a given query. The physical query plan is represented by a graph where each node represents a query fragment, and each link represents inter-worker communication between the execution of two query fragments. Each query fragment is composed by a collection of query operators that can be executed together as one job.

The user can perform three classes of actions on the Physical Query Plan view that will render a new visualization in the performance panel:

- **Empty selection:** by default if no fragments or edges are selected in the Physical Query Plan view, an Fragment Overview visualization is rendered in the performance panel. This visualization displays the aggregate worker utilization over time for each fragment and provides cluster utilization data at a glance.

- **Fragment selection:** when selecting a fragment, a Fragment Execution visualization gets rendered in the performance panel. The Fragment Execution visualization provides aggregate cluster utilization information complemented by per-worker task schedules. When a fragment is selected, the operands inside of the fragment in the Physical Query Plan view are color-coded to allow the user to easily match each task in the per-worker task schedule in the performance window with the corresponding query operator in the browser window.

- **Edge selection:** upon selecting one or more fragment-to-fragment edges, a Worker Communication visualization gets displayed, providing information on inter-worker communication transitioning from one query fragment to the next.

Graph Rendering: We use D3 [3] to render the Physical Query Plan graph, which allows us to support various interactions and transitions. We use GraphViz [4] in the backend to generate graph layout data. GraphViz generates graph layouts that are optimized to minimize area footprint. The layout information is then fed into a D3-based rendering engine which supports various interactions and animations techniques.

Graph Navigation: A query plan can be arbitrarily large. Thus we offer two mechanisms that facilitate exploration of the graph for the user: (1) expanding/reducing fragments and (2) paning. The first mechanism can reduce the size of a graph by a constant factor by collapsing operators that compose a fragment into a single fragment node. The user can click once on a collapsed fragment to expand it, and click once on an expanded fragment to collapse it. Expanding or reducing a node can cause large changes in the graph layout as GraphViz changes the graph layout to minimize overall area. This graph re-shuffling is illustrated in Figure **??** where expanding Fragment 2 causes the layout to change. To address these layout changes, we implemented transitions to allow the user to track the fragments as those get reshuffled. The second mechanism that facilitates exploration of the graph for the user is paning. This feature was implemented in D3 and allows the user to navigate a graph when it doesn't fit inside of the browser panel.

Tooltip Information: **[[write me]]**

3.3.2 *Fragment execution view*

The Fragment Execution view comprises two charts: the utilization chart and the operators chart. The utilization chart (Figure 3) shows how many workers a fragment is running on over time. The chart is used to quickly reveal any patterns in the schedule, like tail latency, long periods of idleness, etc. We implemented *focus+context*[6] using a small brush (at the top of the figure) that allows the user to zoom-in and further analyze a problematic area without loosing context on the entire execution.

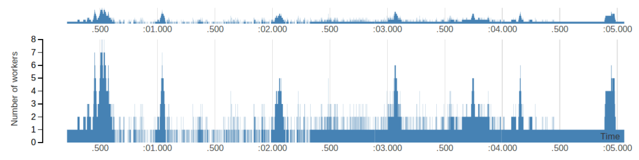


Figure 3: The utilization chart. The user can brush on either the small graph on top or the bottom graph to zoom into a region of interest.

The utilization chart also implements a brush that allows the user to further analyze query execution by displaying the work schedule of each worker in the operators chart (Figure 4). The per-worker schedule displays what operators each worker is executing over time.

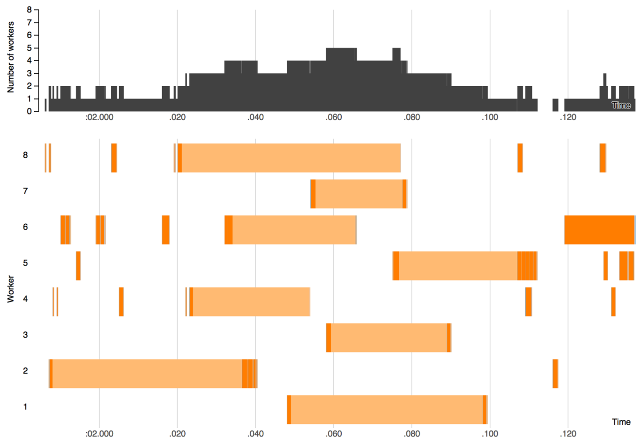


Figure 4: The operators chart.

3.3.3 *Overview over all fragments*

The Fragment Overview shows small multiples of the utilization chart described in the previous section. In addition to identifying execution skew, the user can compare the fragments and find correlations between the execution of different fragments. This view helps the user navigate between fragments and is not intended to offer deep insight into performance issues. Section 4.1 describes a usecase for the Fragment Overview visualization.

3.3.4 *Worker Communication view*

The Worker Communication view consists of two charts: the aggregate communication chart and the time series chart. The aggregate communication chart is a matrix (Figure 9) representing a cluster of N workers. Each cell m, n in the matrix represents the total number of tuples sent by worker m to worker n during the query execution. We chose to use the colors from color brewer[7] and interpolated the colors between the steps using the *CIE L*a*b** interpolation and lightness correction [2]. The colors work in grayscale and can be interpreted by color-blind users.

The matrix chart representing aggregate communication between workers help uncover data skew issues, as discussed in Section 4.

We initially considered having a chord diagram[3] instead of a matrix for the Worker Communication chart. But feedback from our peers convinced us that a chord diagram would suffer from scalability problems. Adding an option for displaying a chord diagram remains part of future work.

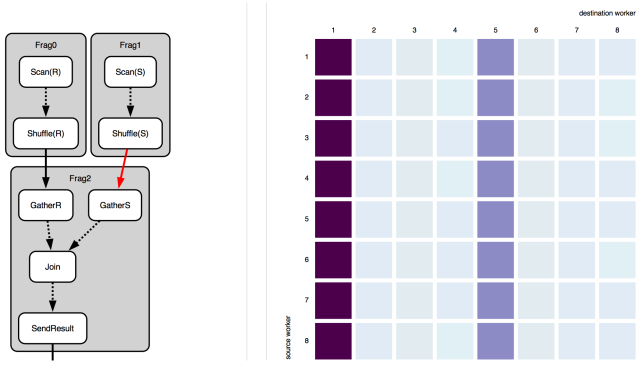


Figure 5: Worker Communication matrix chart. Each cell represents the number of exchanged tuples between a pair of workers.

When the user clicks on a matrix cell, she is able to view the breakdown of the worker pair communication in a time series chart below the matrix. The user can select multiple such cells to compare different time series. Figure 6 shows an example when the user selected two pairs: (8-1) and (7-5). The x-axis shows time in seconds and y-axis shows number of tuples exchanged between the source and destination workers. Just looking at the matrix chart it is evident that there is more data exchanged between the source-destination pair (8-2) than the pair (7-6). But as the time series comparison shows, there is more communication between (7-5) than (8-1) for the last 2.5 seconds of the query execution. This points to interesting patterns as to the rate at which work is done at different workers at different stages of the execution. Distinct points on the lines are highlighted as black dots. Hovering the cursor over a dot shows the information for that data point in a tooltip. Meanwhile the cells in the matrix remain highlighted using the same color used to draw the corresponding time series line. This helps the user keep track of the mapping between the matrix cell and the corresponding time series line. We also provide a 'clear selection' button that erases everything from the time series chart upon clicking. Finally we provide the functionality whereby clicking a row or column label selects every cell in that row or column

[3]`http://bl.ocks.org/mbostock/4062006`

and displays the corresponding time series in the time series chart.

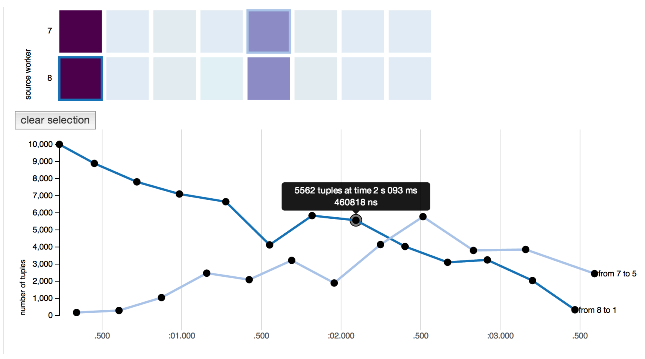


Figure 6: Worker Communication time series chart. The user can select multiple cells in the matrix to compare the corresponding time series.

4. EVALUATION

We used QuiViz to examine real world database queries that ran on Myria. We base our analysis on the different visualizations that were generated by QuiViz and present examples where the QuiViz helped us better identify (1) data skew, (2) execution skew, (3) performance bottlenecks.

We explore a simple join query we performed on one million tuples sampled from a Twitter user graph. This query joins Twitter followers and followees by user ID. The query can be written in SQL as:

```
SELECT *
FROM twitter S, twitter R
WHERE S.follower = R.followee
```

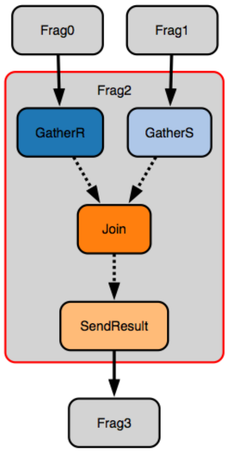


Figure 7: This view shows the Physical Query Plan for a join.

The resulting Physical Query Plan visualization is displayed in Figure 7. The user will navigate the Physical Query Plan visualization in the browser panel in the next case studies to identify different performance issues.

4.1 Case 1: Identifying Performance Bottlenecks using the Fragment Overview

The Fragment Overview view provides the user with a summary of the execution of all fragments on all workers as described in Section 3.3.2. In Figure 8 the user can see for instance that Fragment 2 is irregularly executed on multiple workers. If the graph is not one solid block as it in Fragment 1, 3 and 4, one can assume workers are sleeping. Having workers sleep can be caused by one of two reasons: (1) the workers don't have enough data available to work on (i.e. data dependency problem) or (2) the destination worker's input buffers are full thus forcing the source workers to stall (i.e. back-pressure problem).

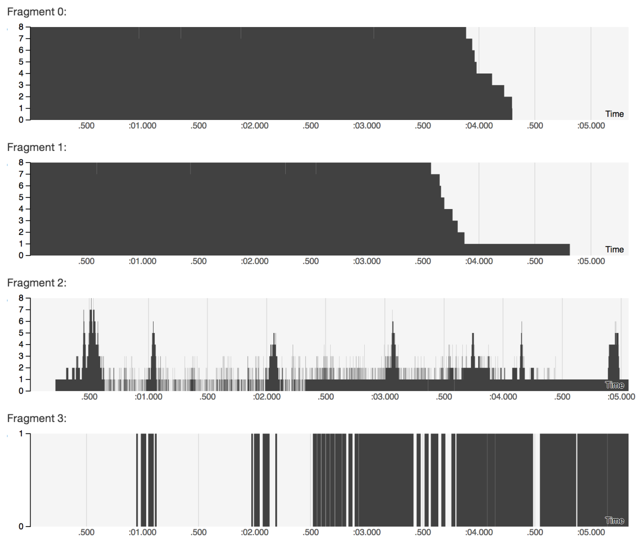


Figure 8: This Fragment Overview shows very irregular usage in fragment 2.

4.2 Case 2: Identifying Data Skew using the Worker Communication View

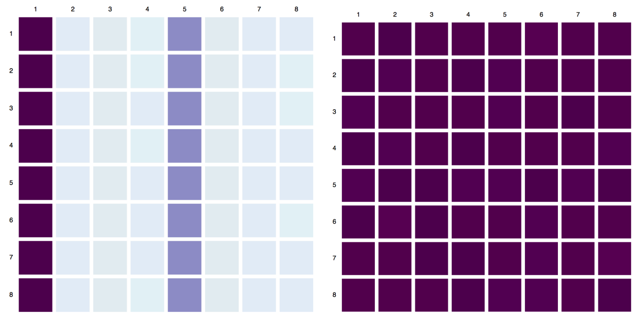


Figure 9: The network view shows a distinctive pattern when there is data skew. Figure 9a shows that most tuples are sent to worker 1 and 5 (origin: row, destination: column).

The Worker Communication visualization at a glance shows how much data was sent between workers by one query fragment to the next. The two Worker Communication views in Figure 9 were obtained by selecting the Fragment0→Fragment2 and Fragment1→Fragment2 edges respectively in the Physical Query Plan view. Figure 9 displays aggregate data communication between workers. In Figure 9b, the amount of data sent from each worker to every other worker is balanced as opposed to Figure 9a where one can see that the volume of data sent to worker 1 and worker 5 dwarfs data sent to the other workers in the system. This apparent data skew could be caused by poor partitioning.

4.3 Case 3: Identifying Execution Skew using the Fragment Execution View

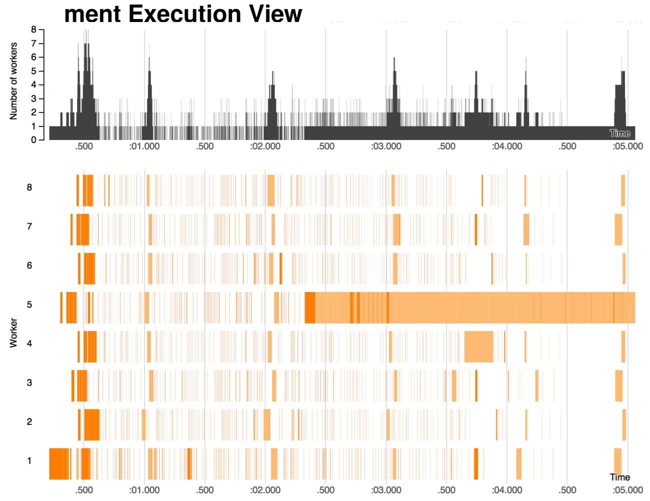


Figure 10: This Fragment Execution view shows heavy execution skew on worker 5.

The Fragment Execution visualization allows the user to dive down into query execution details at the worker level. The Fragment Execution view in Figure 10 was obtained by selecting Fragment2 in the Physical Query Plan view. Fragment2 is the fragment that performs the join. The user can observe at a glance that certain workers produce much more data as a result of the join execution. Particularly, we saw in Case 2 Figure 9a, that tuples sent to Fragment2 from Fragment1 were heavily skewed towards being sent to worker 1 and worker 5. In Figure 10, the user can now see that as a result of the join, worker 5 ends up producing most tuples that have to be written back to disk. The color scheme used for each worker schedules in the Fragment Execution view is color-coded to match the operators in Figure 7. Consequently, one can quickly observe in the Fragment Execution view that worker 5 is busy sending join results from $t = 2.5s$ to $t = 5.0s$, which differs from the rest of the workers. Using similar reasoning, one can conclude that worker 5 presents a performance bottleneck in the execution of the query on Myria. In the short term, the user can modify the way the input data set gets partitioned. Alternatively the user could change the number of workers allocated for the job. In the long term, developers should look at ways to improve data flow to minimize idling on workers that are waiting on input data.

4.4 Discussion

All in all, QuiViz allows the user to get much more insight into query execution than what was available before, i.e. text-based performance logs, and runtimes data. Being able to navigate performance data using a simple visual interface can indeed facilitate the task of a developer who is designing Myria's back end, or a Myria user who is writing complex queries to process large amounts of data. That said, QuiViz can only help the user visualize query execution performance. QuiViz but won't necessarily point the user to what needs to be done to improve query performance. We believe that an experienced user will quickly learn how to identify symptomatic performance execution, and know what steps to take to improve performance as a result. We would have to back this up with a user-study, which could be the object of future work in this direction.

5. CONCLUSION

This paper introduced QuiViz, a visualization tool carefully tailored to help database developers and users to quickly identify common distributed databases performance bottlenecks, such as stragglers, data skew, etc. Our tool has been successfully integrated with the Myria distributed database management system, and has already been used to identify several bugs, such as poor physical query plan optimization, stragglers, poor data partitioning etc. This has led us to believe that the visualization techniques we employed are effective in helping the user quickly narrow down and identify the cause of the performance bottleneck. QuiViz could be easily integrated with another distributed database system, the only requirement being a specific log data format. We explained how we collect logs in Myria and recommend our approach to developers that are porting QuiViz to a new DDBMS.

6. FUTURE WORK

The current implementation focuses on scalability. We managed to some extent to address some scalability issues raised by big query plans, large amount of workers or long running queries. For example the Worker Communication view uses a scalable heatmap to show the communication between all the workers in the system. However, other techniques have to be further implemented to address this issue. One example would be a more interactive query plan graph, which the users can zoom into. Also, by querying the performance log data on the backend, the Fragment Execution view should only download and render what the user can see.

Besides performance improvements, we plan to integrate X-trace and its visualization to offer the orthogonal view that allows users to trace how tuple batches flow through the operators and between operators. Combined with the visualizations described in this paper, it will form a powerful debugging tool that will help users better understand and improve query execution.

7. REFERENCES

1. Myria. `http://myria.cs.washington.edu`.

2. G. Aisch. Mastering multi-hued color scales with chroma.js, 9 2013. `https://vis4.net/blog/posts/mastering-multi-hued-color-scales/`.

3. M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

4. J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies. Graphviz open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

5. R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

6. G. W. Furnas. *Generalized fisheye views*, volume 17. ACM, 1986.

7. M. Harrower and C. A. Brewer. Colorbrewer. org: An online tool for selecting colour schemes for maps. *The Cartographic Journal*, 40(1):27–37, 2003.

8. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

9. A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The sdss skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 570–581. ACM, 2002.

10. Twitter. Zipkin, 2012. `http://twitter.github.io/zipkin/`.

11. Twitter. Ambrose, 2013. `http://github.com/twitter/ambrose/`.