Query Performance Visualization

Tim van Elsloo (teo400)
University of Amsterdam
tim.van@elsl.ooo

Thomas van der Ham (thm280)
University of Amsterdam
thomasvanderham@outlook.com

ABSTRACT

We present novel visualization software that aids in profiling queries, identifying bottlenecks and equalizing workload distribution. We used our software to visualize and compare query performance on Apache Spark and Apache Pig for two widely used benchmarks: TPC-H and JCC-H. Our software is agnostic to database software and can easily be adapted to support other software as well.

1. INTRODUCTION

With vast amounts of unstructured data, there is a growing need for big data query engines. One of the first distributed query engines was MapReduce[4]. Hadoop[8], a distributed file system, also supports MapReduce queries out of the box. However, a downside of using low level software like MapReduce is having to write specific map and reduce functions for each query instead of writing in an intuitive, high-level language like SQL. For that purpose, big data query engines have started to include options to use higher level languages. Some engines have implemented SQL support or come with their own language. This enables developers to quickly develop queries that can run in parallel on distributed systems. Another advantage of these high level languages is that they can offer a set of common optimizations to improve query performance.

In this paper, we perform a comparison on two different big data engines: Apache Spark[9] and Apache Pig[6]. With Spark, we specifically inspect its Spark SQL[1] capabilities. Pig does not support SQL, but its own Pig Latin language. While Pig Latin is not a high level language, it certainly is not as low level as MapReduce. We compare performance with our own visualization software, that was written as part of this contribution. Both engines are compared on widely used benchmarks: TPC-H and JCC-H. Our research goal is to use these visualizations to identify the differences between TPC-H and JCC-H[2] on each of the queries in the benchmark.

In sec. 2, we will first cover the query engines that we have selected for our performance comparison, as well as quickly introduce the two benchmarks and explain how they relate to each other. Then, in sec. 3 we show how we load vast amounts of data into Hadoop in an efficient, parallel way. Next, in sec. 4 we present our novel visualization software. In sec. 5, we use our software on 3 case studies: selected queries from the benchmark. Finally, we conclude this paper in sec. 6 with a brief discussion on our results.

1.1 Related work

In a small performance comparison between Shark, Impala and Spark SQL, [1] have shown that the fastest engine depends on the selectiveness of the query. Spark SQL performs better on less selective queries and Impala performs better on more selective queries. The old version of Spark SQL named Shark is outperformed by both of them.

[5] also showed that the performance of Impala is on par with Spark SQL and they added a comparison with Hive which performed similarly. They showed that the memory usage of Impala was higher than the memory usage of Hive and Spark. The formatting of the file has a major impact and compression could result in a speedup if i/o causes a bottleneck.

Software similar to ours has also written before, such as Twitter Ambrose[10]. However, to our knowledge we are the first to publish software that can both visualize queries in great detail, as well as show workload distributions (detail per node, operation).

2. PRELIMINARIES

Before we present our own work, we will first briefly discuss our setup. Our experiments observe four metrics: time, records, input / output (i/o) and memory usage. We quickly noticed that the time metric is heavily influenced by environment variables beyond our control (mainly the workload that is submitted to the supercomputer by other groups). Fortunately, most of these jobs are long-running jobs. Therefore, we find it beneficial to visualize relative times (as percentages of the total time of a query) rather than volatile absolute times. In addition, observing the other metrics adds accuracy to our query profiles than only looking at time.

This section will continue with a brief explanation on our query engines selection in sec. 2.1 and will conclude with the benchmarks that we have compared these engines on in sec. 2.2.

2.1 Query Engines

As part of our experiments, we validate our software on two different database engines. We have chosen Apache Spark and Apache Pig because both were pre-installed on the supercomputer that we have been given access to. In this section, we first cover Apache Spark, which comes with an SQL layer to enable rapid ad-hoc querying. Then, we will discuss Apache Pig, a different engine that does not support SQL and instead compiles programs in Pig Latin directly to MapReduce tasks.

2.1.1 Apache Spark

Apache Spark is a scalable big data processing engine that simplifies the task of processing big data by supporting several high level languages like Python, R and SQL[11]. Spark uses existing filesystems like Hadoop or Amazon S3. Spark is fault tolerant, allows for streaming data and provides low latency computing. Spark uses in-memory processing when possible rather than writing each intermediate step to the distributed file system (which is what a sequence of MapReduce jobs does). According to Apache this makes Spark ten to hundred times faster than MapReduce.

Spark SQL leverages the benefits of relational processing, while allowing users to use complex analytical libraries in Spark[1]. It provides support for features that are designed for big data analysis such as semi-structured data, query federation and special data types for machine learning. It also allows users to write complex pipelines in which they can mix relational and complex analytics which are automatically optimized using its optimizer. The optimizer is extendable using Scala, which allows users to add new optimization rules, new data types and new data sources.

2.1.2 Apache Pig

In addition to Spark, we have also run our experiments on Apache Pig: another big data query engine. Pig does not support SQL, instead we used a set of equivalent queries for TPC-H (sec. 2.2.1) in Pig Latin, written by the developers of Pig. We have manually updated these queries with the parameters generated by JCC-H (sec. 2.2.2).

Unlike Spark, Pig compiles programs to a sequence of MapReduce jobs. The output of those jobs is stored in the distributed file system. Pig also offers a variety of query optimization heuristics.

Unfortunately, the statistics that Pig outputs (even in verbose modes), are less detailed than those that we can gather from Spark. Therefore, this paper will have a slight focus on Spark. In our conclusions (sec. 6) we propose directions for future work to overcome this problem.

2.2 Benchmarks

In order to validate the usefulness of our visualization program, we use it to profile queries from 2 widely used database benchmarks: TCP-H and JCC-H. The latter is an improvement upon the former: it is a better approximation of real world workloads. In this section, we will first take a look at the original TPC-H benchmark in sec. 2.2.1 and continue by comparing the newer and better JCC-H benchmark in sec. 2.2.2. Both benchmarks support a scale factor parameter that determines the size of the generated synthetic datasets.

2.2.1 TPC-H

The TPC-H benchmark approximates an industry that manages, sells or distributes products worldwide[7]. This benchmark models the decision support area of a business, where trends and refined data are produced in order to make good business decisions. TPC-H exists of two parts: data generation and query generation.

The data generation can be executed on different scale factors which can run compliantly when specific values are used in the range of one through one hundred thousand. Where one stands for one gigabyte of data. The data that is generated can be split in multiple parts, so it can be distributed over multiple processes. The generated data exists of eight tables which are all heavily connected through foreign key relations. According to the guidelines of the benchmark the tables created based on the data may not rely on any knowledge of the data except for minimum and maximum values of the fields.

The benchmark defines a set of 22 query templates that simulate the type of ad-hoc queries a real world business would use. The queries are designed to include a diverse set operations and constraints, generate a big CPU and disk load and should be more complex than most online transactions. The queries address some real world choking points[3] that are found when handling big datasets. The queries are generated based on an random number in such a way that the performance for a query with a different substitution should be comparable.

2.2.2 JCC-H

The JCC-H benchmark is a drop-in replacement for TPC-H generator which should produce a dataset which is more realistic[2]. This is achieved by adding Join-Cross-Correlations and skew to the dataset and the queries. A Join-Cross-Correlation is values occurring in tuples from one table which can influence behavior of operations on data from other tables when used in a operation that joins these tables. This is added to the benchmark by generating data that is better correlated than data generated by TPC-H and also using the correlated data in the generated queries. Some join optimizers try to optimize for correlation, but some fail to estimate the cardinality which makes it interesting to test.

Skew is the asymmetrical distribution of data, which occurs in the real-world but it does not occur in the data generated by TPC-H. The distribution of the data generated by TPC-H is uniform. JCC-H adds skew to the generated data to make the dataset and references the skewed data in the queries to give a better reflection of real-world performance. The skew is introduced by letting each referencing table have twenty-five percent of all it's tuples refer to only a small part of the foreign keys. In the real-world this is a well known issue which affects the partitioning of the data.

3. DATA INGRESS

We use two synthetic database benchmarks (sec. 2.2) and both generate 1 terabyte of data. In order to load such vast amounts of data into Hadoop in the first place, we need to parallelize the ingress process. In addition, when querying data, we want to be able to distribute the workload on all available nodes in our cluster. This requires the use of table partitioning. In this section, we first explain how we achieve parallel loading and continue with our table partitioning strategy.

3.1 Parallel Loading

In order to optimize loading our datasets into the Hadoop distributed file system, we exploit the fact that our benchmark is synthetic. Both programs support options to generate partial data. With that in mind, we distribute the programs over all nodes in the Hadoop cluster and run those programs with slightly different arguments on each node. These arguments determine the part of the data that each node generates, and the location that it stores the generated data onto (sec. 3.2).

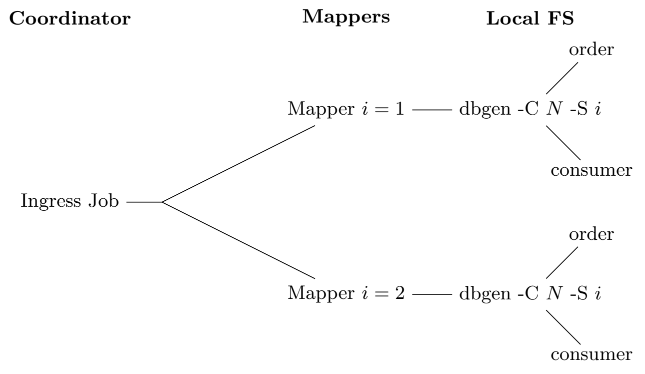


Figure 1: Overview of our parallel loading MapReduce job. Each mapper is assigned an ID (i), knows the total number of nodes (N) and only generates $[\frac{i}{N}, \frac{i+1}{N}]$ of the data.

In fig. 1, we show all of the steps that are taken to load the data in parallel onto the file system. Our data loader program is wrapped as a MapReduce task, even though there is no reduce-stage. It starts by generating a sequence of indices (one for each node) and mapping those indices to nodes. We override the default Hadoop MR configuration to ensure that each node is assigned one, and only one, index. The Hadoop scheduler then takes over and distributes our task to all available nodes. Each node generates its assigned part of the data and temporarily scores the results on its local file system. Finally, we transmit the generated files from the local file system to the distributed file system. In the next subsection, we will explain how the approach we take to transfer those files affects table partitioning.

3.2 Table Partitioning

On Hadoop, each table is stored as a separate folder in the file system. Each of the nodes that we use to load data into Hadoop, writes to its own file. Therefore, the parallelism of our data ingress process also determine the level of table partitioning.

The advantages of partitioning each table in several files, is that upon querying the dataset, each worker node can autonomously process a distinct subset of the data.

In fig. 2, we show a detailed overview of how table partitioning works. The figure shows that during ingress, each table is stored as a folder of several distinct parts of all data. When querying the data, each worker only reads from a few parts of the data (depending on the ratio between ingress parallelism and query parallelism).

It takes approximately 2 minutes to generate and store 30gb of data (SF = 30) and approximately 14 minutes to generate and store 1tb of data (SF = 1000), with parallelism of 100 nodes.

4. PERFORMANCE VISUALIZATION

We present a novel approach to visualizing query performance by drawing inspiration from modern user interfaces for geographical maps. Our query visualizer software turns queries into abstract syntax trees (AST) and treats edges as a heat map of various user-selectable metrics: time, records, input / output (i/o) and memory usage.

In this section, we first present our query visualizer. We then show a extension that we built on top of the query visu-

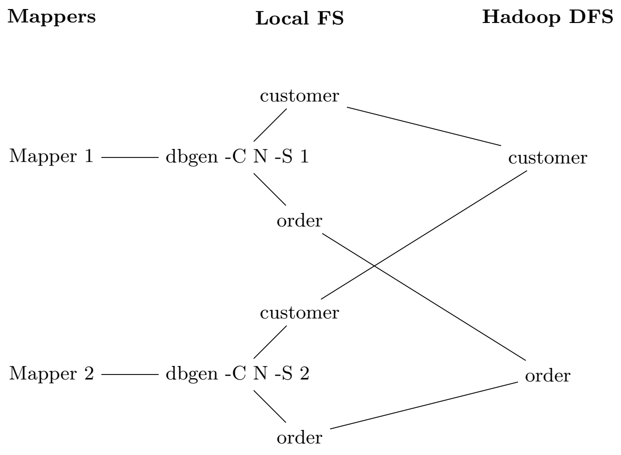


Figure 2: Overview of how we partition the tables that each individual node generates. Note that each line to the tables represents a distinct file in the Hadoop distributed file system.

alizer to show workload distribution, which simplifies identifying bottleneck subqueries. Finally, we present a timeline that is added to the visualization to show the start and end times of stages (groups of operations) that are part of the query plan.

4.1 Query Visualizer

Our query visualizer shows an abstract syntax tree (AST) of the physical plan that is generated by each database engine. The physical plan is closely related to the original query and describes an order of execution of separate subqueries. In Spark terminology, these execution units are called *stages* and in Pig, the same is called *jobs*. Pig compiles a query into a sequence of MapReduce jobs. An unfortunate consequence is that it does not offer monitoring at a lower level than jobs[1]. In contrast, Spark is able to provide much more detailed monitoring statistics.

In both cases, we spit through the log files that both engines generate to find metrics that we can use for our visualization. For Pig, we also query the Hadoop REST API to download and parse additional logs for each node that our MapReduce jobs run on. We provide software with our submission that automatically downloads and parses these logs.

In fig. 3, we show the end result of our query visualizer. The color of each edge represents its fraction of time (or one of the other three metrics): thick red edges are time expensive operations, thin blue edges are very fast.

Our software contains a sidebar with settings that can be used to select one of the four supported metrics (time, records, disk i/o, memory usage).

In order to obtain a physical query plan from Pig, we run `EXPLAIN -script Q1.pig`. We rebuild this physical plan tree into the representation that is shown in our visualization. For Spark, we use the events `json` log file and parse the `SparkListenerSQLExecutionStart` event to obtain a Spark plan tree.

[1]Except for user-defined functions: `pig.udf.profile`.

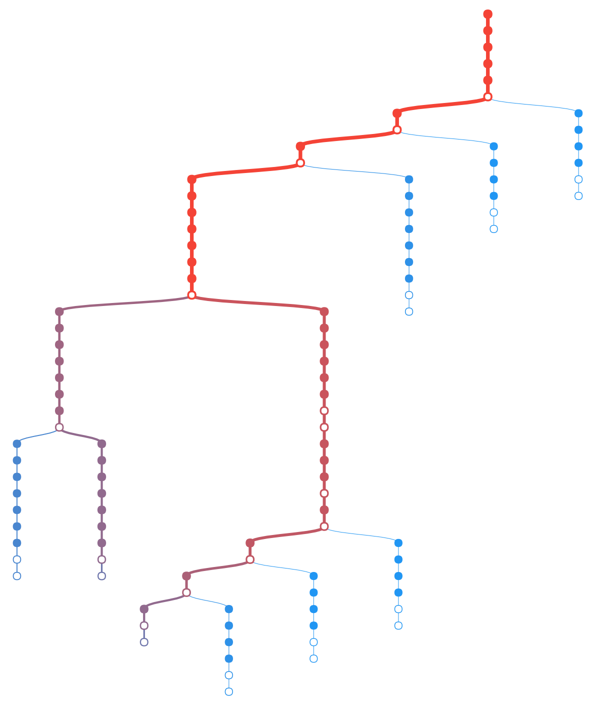


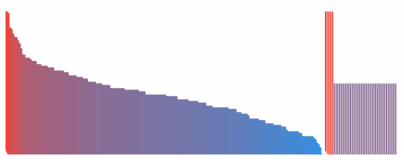
Figure 3: Visualization of Q2 on Spark. Selected metric is the number of records. Node names (e.g. `filter` and `load csv`) are omitted in this figure for simplicity but are included in our software.

4.2 Workload Distribution

In addition to the query visualizer, we also built an extension that visualizes the workload distribution for each node in the AST (depending on the metric that you select). Nodes for which workload distribution information is available, are indicated with a white circular fill. For example, in fig. 3, all nodes that have 2 children have workload distribution information.

Workload distribution visualization simplifies identifying bottlenecks: data processing steps that cannot easily be parallelized. As an example, in fig. 4 we show two workload distributions for the same node: one is part of the TPC-H benchmark and the other is part of the JCC-H benchmark.

Note that we again support four different metrics and these often do not correlate: for example, using more memory might speed up the program.



(a) TPC-H (b) JCC-H

Figure 4: Visualization of workload distribution of the same operation, on TPC-H and JCC-H. Each bar represents the relative work that a node performs. Red indicates most of the work, blue indicates least of the work. Nodes are sorted by their share of work in descending order.

4.3 Timeline

We have also added a small timeline of stages at the bottom of our visualization window. The timeline shows starting points end ending points (in relative time) of all of the stages. Note that in rare cases, a few of the stages that appear in the timeline may not have counterparts in the visualized tree (because those stages do not relate to any nodes in the tree). Also, Spark in particular tends to skip stages if it determines at run-time that it already has the results of a those stages as part of some other stage.

5. EXPERIMENTS

One of the most common applications of our software is to use it for identifying skewed workload distribution. Systems based on MapReduce usually work on the principle that each key takes an equal amount of time to map. In real world data, that is hardly ever the case. Our second benchmark JCC-H (sec. 2.2.2) specifically aims to replicate real world data distributions in order to better reflect actual performance. It has already been shown that both Hive and VectorH query engines have more difficulty with the JCC-H than TPC-H[2]. Therefore, as part of our research goal, we show that our program can identify skewed workload distribution in Spark and Pig as well by comparing the results of running the TPC-H benchmark and JCC-H benchmark.

Our experiments consist of comparing our visualizations for 3 of the most complex queries (in terms of number of distinct stages) between TPC-H and JCC-H and using our software to identify potential bottlenecks that are caused by the skewness of the latter benchmark. In addition, we compare our findings with the design goals of each query as written by the authors of JCC-H[2]. This selection was based on fig. 3 and 4 of that paper: it shows that query 7 and query 9 are significantly more difficult with skew. We have chosen 18 because it shows that Spark is able to reduce the effect of skew by reorganizing the query plan.

In addition to the experiments below, we have also confirmed (as a means of grounding our experiments) that our visualizations of Q1 are identical (regardless of benchmark) and no abnormalities can be detected, because both benchmarks produce the same parameter sets.

Overall, as written in our preliminaries, it is hard to measure the absolute runtime of each run. In our experiments, Spark usually was faster than Pig, although this can be attributed to many external factors (most notably: workload by other project groups). Therefore, we will be looking exclusively at relative time, io and memory utilization and focus on the workload distribution graphs that are generated by our software.

5.1 Query 7

With query 7, we observe severe skewness in the workload distribution with respect to the number of records when joining customers and nations with the rest of the query (`BroadcastHashJoin` and `SortMergeJoin`). What this means, is that the workload for each nation and customer is not uniform (some nations and customers demand much more work). This ultimately slows down the query because there are many nodes that are idling and only a few that are very busy. We only observe this skewness in the JCC-benchmark (similar to fig. 4) and believe that this is inline with the assertions made by the original authors. In appendix we have added the entire visualization window, with red circular annotations indicating bottlenecks we have detected using our software.

Note that we can see Spark optimize this query by reordering the timeline: the order of execution of each stage. In particular, stage 4 is started almost immediately for TPC-H, whereas its start is delayed until stage 3 is done for JCC-H. Stage 4 loads customers and stage 3 loads line-items.

5.2 Query 9

The updated query 9 introduces various bottlenecks. In these cases, many nodes are idle while only a few are doing the work. We have annotated operations with non-uniform workload distribution in appendix . It is important to note that even though the workload distribution is not uniform, it does not seem to have a significant impact on the overall performance because Spark spends most of the time at the last few nodes (which are visualized on the north side). These nodes however, are not impacted by the bottlenecks that are introduced.

We confirm this observation by noting that Spark does not significantly change the order of execution (which was the case with Q7), nor changes the stages to optimize a specific path in the AST (which we will see with Q18).

On Pig, we do not observe a bottleneck: all workload distributions are nearly identical to their normal (unskewed) counterparts.

5.3 Query 18

When analyzing query 18, we can see that Spark does considerable effort to rearrange its query plan to optimize for the updated parameters. We can also observe that memory usage is significantly different with JCC-H than TPC-H, even though the time metric doesn't show significant difference. We believe that this is because of optimizations that specifically accommodate these types of skewed joins. For example, Pig *can* allocate memory in the reducer stage to do the heavy lifting that is required for skewed joins. We believe Spark does something similar, which explains why there is a higher memory use but not a significant change in time.

Ironically, Pig does not handle the skewed query 18 well. The resulting workload distributions (both with respect to time and memory) are significantly non-uniform. Fig. 5 shows the workload distribution of the bottleneck we have identified. We can see that most nodes running TPC-H (left) are responsible for the same share of work, whereas only a few nodes running JCC-H (right) are responsible for most of the work. Note that red bars are not necessarily a bad sign: they represent relative utilization of each node. Ideally, a query shows a (near) uniform workload distribution.

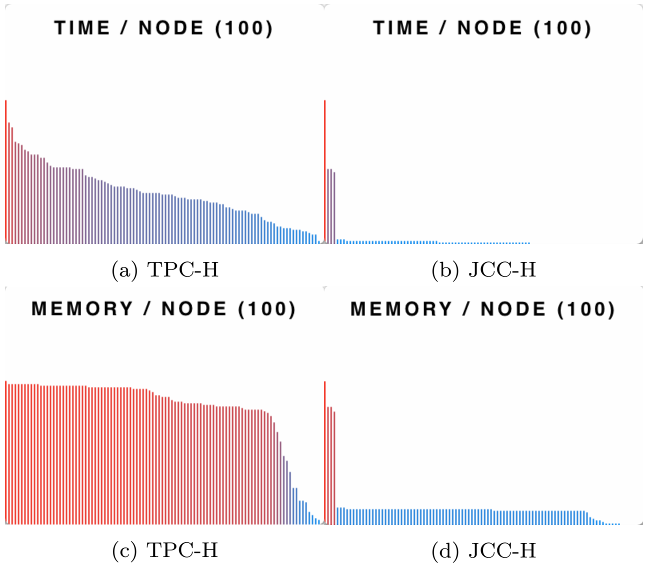


Figure 5: Visualization of workload distribution in Pig of the bottleneck in query 18 that is introduced in query 18, on TPC-H and JCC-H. Appendix C, fig. 9 shows the location of the bottleneck.

6. CONCLUSIONS

We have built an advanced query profile visualizer that supports both Spark and Pig. It performs better on Spark because we have access to statistics in more detail. Our experiments, consisting of 3 case studies on complex queries from the benchmarks, show that our visualizer can be used to identify bottlenecks and performance regressions in queries in great detail. Most of the results and conclusions that are obtained as part of our experiments, align with the goals of the authors of JCC-H[2]. We can also observe that query engines optimize the skewed queries and generate a different execution plan, with a different timeline (sec. 5.1) and stage reorganization (sec. 5.3). In our experiments, we have observed that Spark and Pig use different optimization strategies, with varying success. Specifically, Spark is unaffected by the skewness that is added in query 18 and Pig is unaffected by the skewness added in query 9.

We believe that our software contribution improves over earlier work because it greatly simplifies finding causes of performance regressions and non-uniformly distributed workloads in distributed environments such as Hadoop. We have been able to confirm that it correctly identifies skewness introduced by the JCC-H benchmark.

6.1 Future Work

While we have run our experiments on both Spark and Pig, Spark logs are more granular and contain more metrics than Pig logs. Our approach should serve as a proof of concept and an effort can be mode to integrate the database-specific parts of our work directly into each particular database. This would allow for more precise profiling, as well as a stable integration that does not easily break when updating the database engine to a newer version (e.g. in case logs change).

In addition, future research could focus on simplifying the visualizer even more. Our current visualizer is based on the physical plan, which is often closely related to the actual user query. However, it would be even better if the visualizer could relate parts of the physical plan to the original query. This is a difficult problem because our profiler has to take into account various optimizations that query engines perform: all of these optimizations cause further convergence of the physical plan with respect to the original query.

Finally, future work could be directed into measuring the effects of the introduced bottlenecks in absolute time. This would require a sterile setting, in which all external factors (such as external workload) can be adjusted.

6.2 Deliverables

Our assignment submission contains a) all code that is related to data ingress (sec. 3), b) all code that is required to run our software on SURFsara (sec. 4), c) all queries for TPC-H and JCC-H in both Spark SQL and Pig Latin query languages and d) all output of our software on all queries on TPC-H and JCC-H (sec. 5). Our code is hosted on Github: `https://github.com/peterboncz/lsde2017-group17/tree/p2`. All of our visualizations are stored as `html` files, archived in a `zip` archive and directly attached to our project report submission.

7. REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[2] Peter A Boncz, Angelos-Christos Anatiotis, and Steffn Klabe. Jcc-h: adding join crossing correlations with skew to tpc-h. 2017.

[3] Peter A Boncz, Thomas Neumann, and Orri Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, pages 61–76. Springer, 2013.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Xiaopeng Li and Wenli Zhou. Performance comparison of hive, impala and spark sql. In *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2015 7th International Conference on*, volume 1, pages 418–423. IEEE, 2015.

[6] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[7] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[8] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

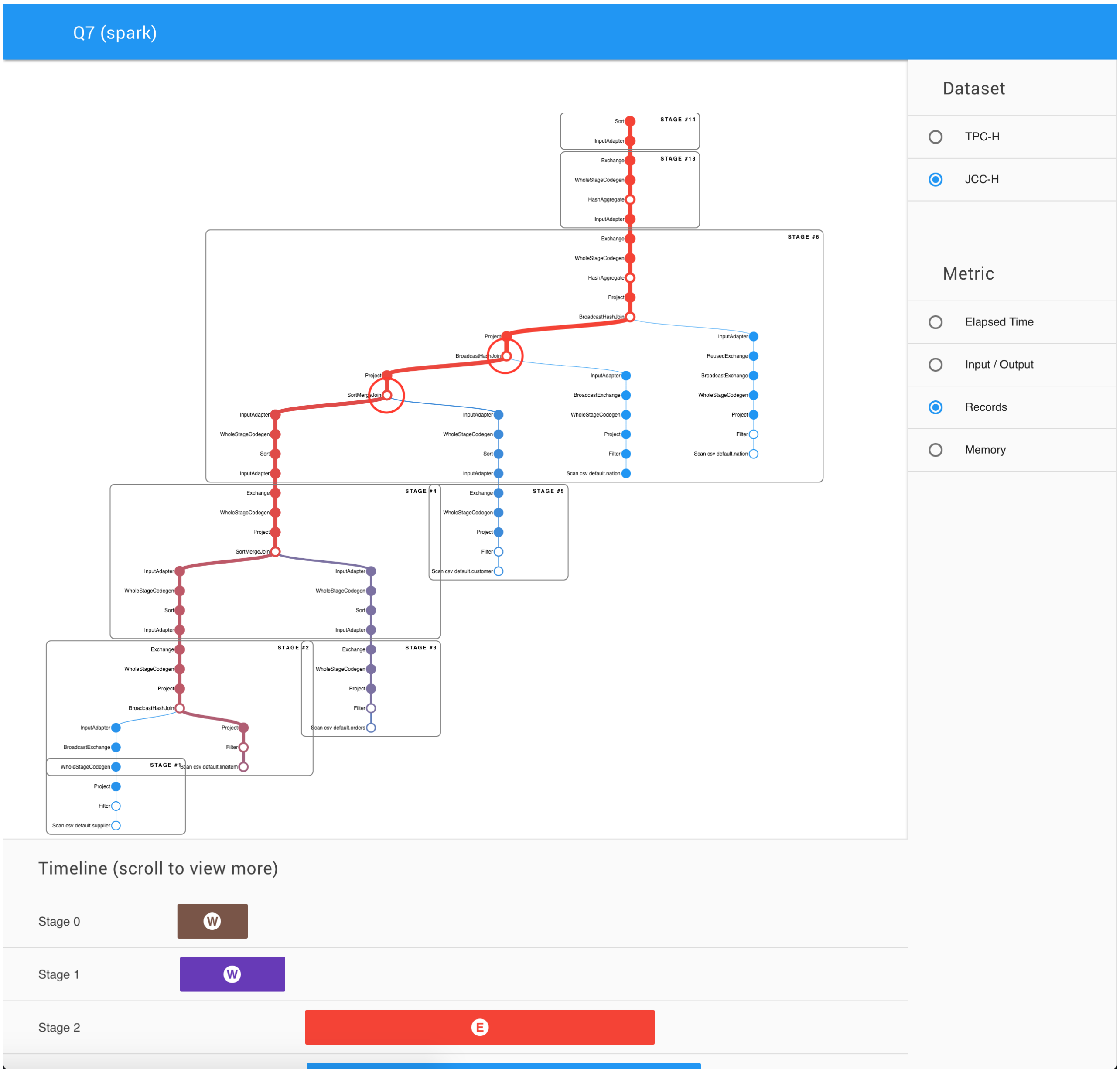
[9] Apache Spark. Apache spark: Lightning-fast cluster computing, 2016.

[10] Twitter. twitter/ambrose, Jun 2016.

[11] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

APPENDIX

A. CASE-STUDY: QUERY 7


Q7 (spark)

Dataset

TPC-H [✓ JCC-H]

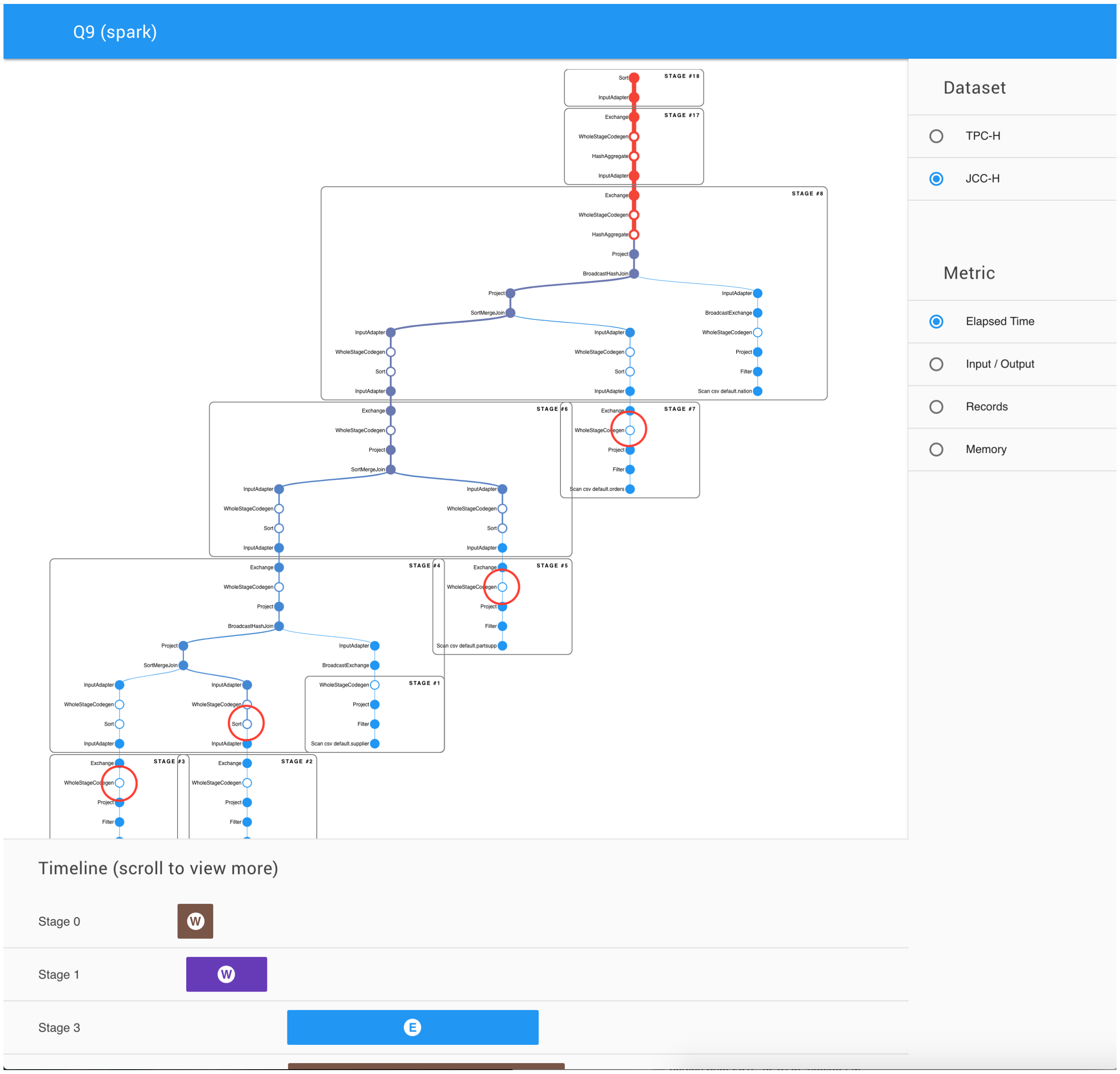
Metric

Elapsed Time / Input / Output / [✓ Records] / Memory

Timeline (scroll to view more)

Figure 6: Interface of a performance report generated by our program for Q7 on Spark (JCC-H, metric is records). This `html` file can be found in `visualizations/spark/Q7.html`.

B. CASE-STUDY: QUERY 9


Q9 (spark)

Dataset

TPC-H [✓ JCC-H]

Metric

[✓ Elapsed Time] Input / Output Records Memory

Timeline (scroll to view more)

Figure 7: Interface of a performance report generated by our program for Q9 on Spark (JCC-H, metric is time). This `html` file can be found in `visualizations/spark/Q9.html`.

C. CASE-STUDY: QUERY 18

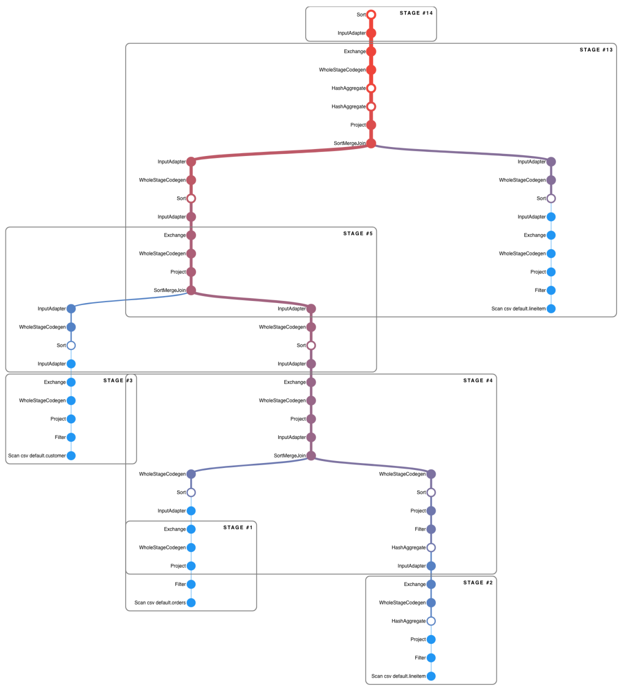


Figure 8: Interface of a performance report generated by our program for Q18 on Spark (TPC-H, metric is memory). This `html` file can be found in `visualizations/spark/Q18.html`.

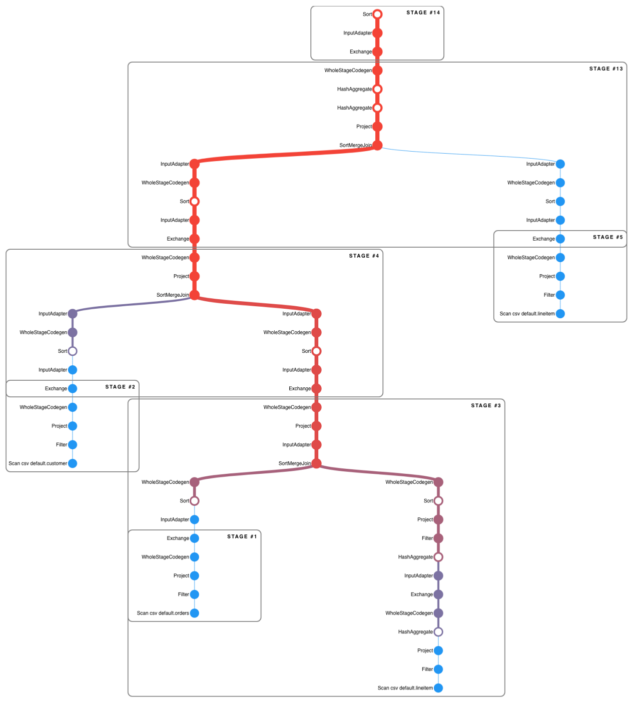


Figure 10: Interface of a performance report generated by our program for Q18 on Spark (JCC-H, metric is memory). This `html` file can be found in `visualizations/spark/Q18.html`.

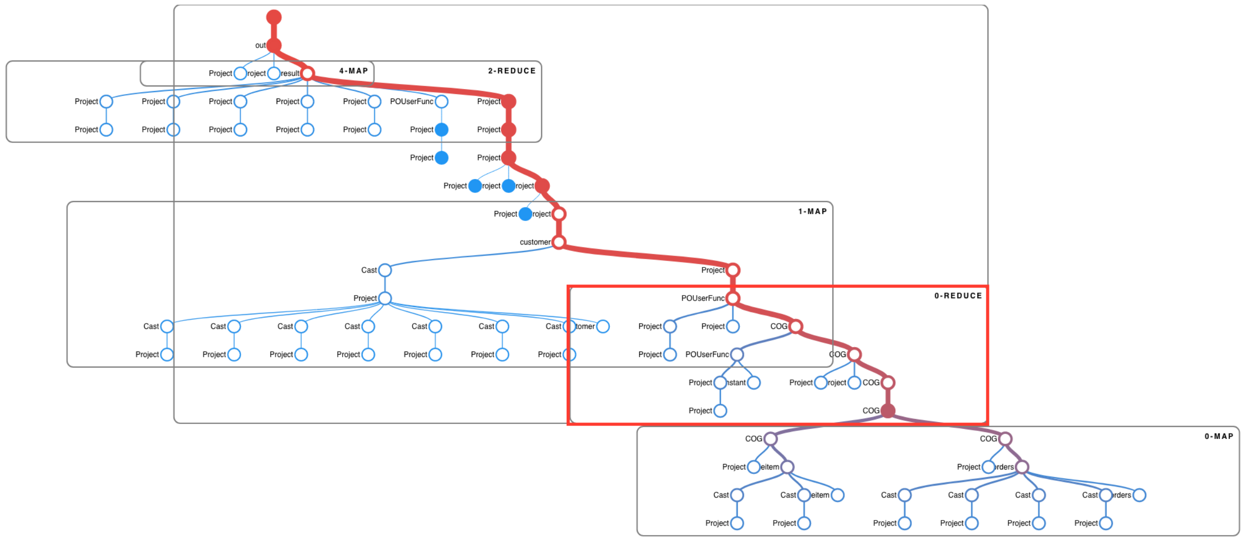


Figure 9: Interface of a performance report generated by our program for Q18 on Pig (JCC-H, metric is time). The rectangular area shows the bottleneck that causes a highly skewed workload distribution. This `html` file can be found in `visualizations/pig/Q18.html`.