QEVIS: Multi-grained Visualization of Distributed Query Execution

Qiaomu Shen, Zhengxin You, Xiao Yan, Chaozu Zhang, Ke Xu, Dan Zeng, Jianbin Qin, Bo Tang

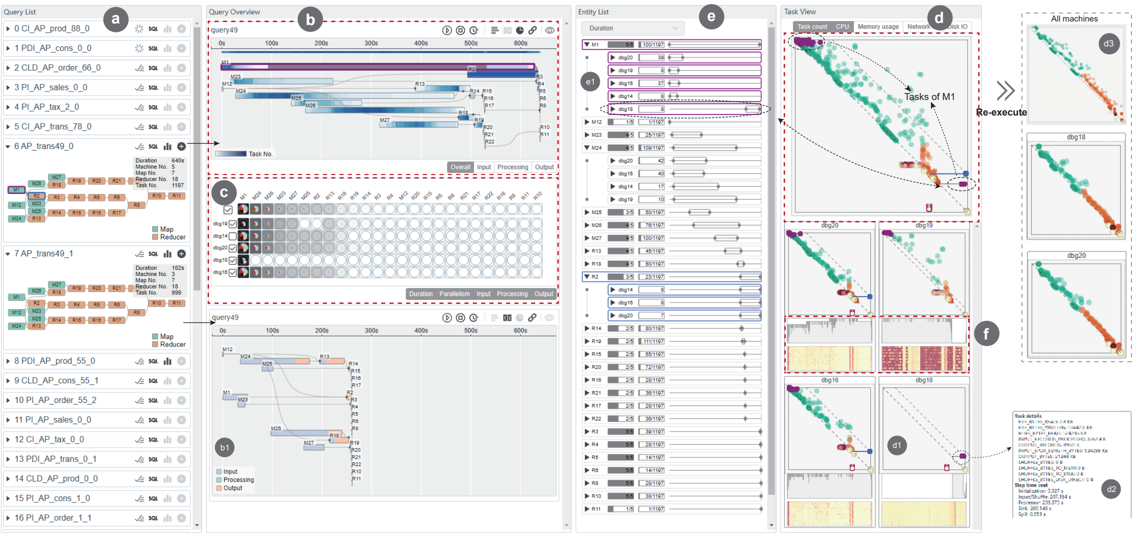


Fig. 1: The UI of QEVIS consists of multiple complementary views. (a) *Query list* displays all queries and enables query selection for analysis. (b) *Job view* depicts the execution progress of the jobs in a query and their dependencies for an overview of query execution. (c) *Performance matrix* reveals anomaly degrees of jobs and machines, facilitating the rapid identification of execution anomalies. To reason about the anomalies: (d) *task view* plots the distribution of atomic tasks. (e) *entity list* reports detailed task execution statistics. (f) *Profiling view* presents hardware status. These coordinated views collectively enhance the exploration of query execution dynamics.

Abstract—Distributed query processing systems such as Apache Hive and Spark are widely-used in many organizations for large-scale data analytics. Analyzing and understanding the query execution process of these systems are daily routines for engineers and crucial for identifying performance problems, optimizing system configurations, and rectifying errors. However, existing visualization tools for distributed query execution are insufficient because (i) most of them (if not all) do not provide fine-grained visualization (i.e., the atomic task level), which can be crucial for understanding query performance and reasoning about the underlying execution anomalies, and (ii) they do not support proper linkages between system status and query execution, which makes it difficult to identify the causes of execution problems. To tackle these limitations, we propose QEVIS, which visualizes distributed query execution process with multiple views that focus on different granularities and complement each other. Specifically, we first devise a query logical plan layout algorithm to visualize the overall query execution progress compactly and clearly. We then propose two novel scoring methods to summarize the anomaly degrees of the jobs and machines during query execution, and visualize the anomaly scores intuitively, which allow users to easily identify the components that are worth paying attention to. Moreover, we devise a scatter plot-based *task view* to show a massive number of atomic tasks, where task distribution patterns are informative for execution problems. We also equip QEVIS with a suite of auxiliary views and interaction methods to support easy and effective cross-view exploration, which makes it convenient to track the causes of execution problems. QEVIS has been used in the production environment of our industry partner, and we present three use cases from real-world applications and user interview to demonstrate its effectiveness. QEVIS is open-source at https://github.com/DBGroup-SUSTech/QEVIS.

Index Terms—visual analytics system, distributed query execution, performance analysis

1 Introduction

Distributed query processing systems such as Apache Hive [49], Spark [2], and Flink [15] support scalable data analytics on many machines [49,50]. With a SQL-like interface on top of parallel data processing frameworks (e.g., MapReduce [21] or Tez [42]), these systems enable users to run queries with SQL semantics instead of implementing low-level parallel computing programs. For engineers and developers working with these systems, analyzing and optimizing distributed query execution are daily routines. Frequently asked questions include "*Where does the query execution time go?*", "*What is the performance bottleneck of the executed query?*", and "*Why does the query run slower than expected?*". Answering these questions requires a comprehensive understanding of the query execution process, thus, it is challenging even for experienced engineers. This is caused by the inherent complexity of distributed query execution, e.g., there are many parallel tasks (i.e., the atomic unit of query execution), and machine status and

- *Qiaomu Shen, Dan Zeng, and Bo Tang are with Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology. E-mail: {shenqm, zengd, tangb3}@sustech.edu.cn.*
- *Zhengxin You, Xiao Yan, Chaozu Zhang and Bo Tang are with Department of Computer Science and Engineering, Southern University of Science and Technology. E-mail: {12250078@mail.,yanx@, 12132372@mail.}sustech.edu.cn.*
- *Ke Xu is with Huawei Technologies Co., Ltd. E-mail: xuke81@huawei.com.*
- *Jianbin Qin is with Shenzhen Institute of Computing Sciences, Shenzhen University. E-mail: qinjianbin@szu.edu.cn.*
- *Dr. Bo Tang is the corresponding author.*

Manuscript received 31 March 2023; revised 1 July 2023; accepted 8 August 2023. Date of publication 26 October 2023; date of current version 21 December 2023. Digital Object Identifier no. 10.1109/TVCG.2023.3326930

concurrent programs can influence task execution in subtle ways.

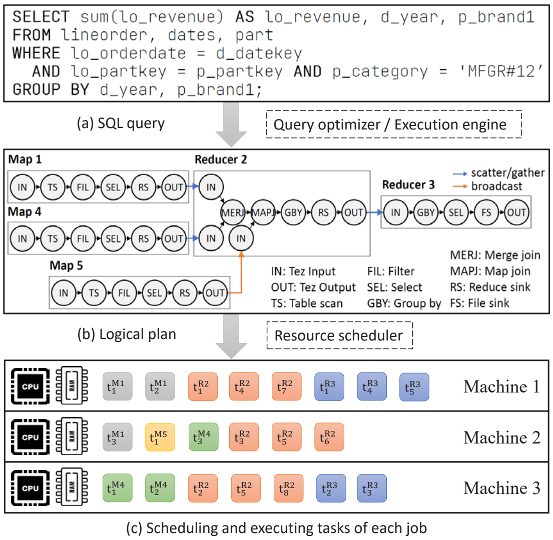
As visualizations allow intuitive interpretation, many visual analytics tools have been developed to assist the understanding of query execution, e.g., Tez UI [9], Spark UI [8], and Dr. Elephant [4]. However, the limitations of these tools are two-fold. (i) Visualization granularity: most of them only offer a coarse-grained visualization of the query execution (e.g., the job level) but fine-grained visualization at the atomic task level is essential for purposes such as tracking deadlocks and detecting data skew. (ii) Tool usability: they provide limited interactions among different views, and thus users have to frequently switch and speculate the linkage among the views. The limitations make it difficult to identify the root causes of abnormal execution behaviors efficiently.

To overcome them, we propose QEVIS, a visual analytics system that allows an in-depth understanding of distributed query execution process. QEVIS encompasses a suite of coordinated views that complement each other by visualizing query execution at different granularities. Specifically, the *job view* displays the execution process of the Map/Reducer jobs in the query and their dependencies as an overview. A new temporal directed acyclic graph (TDAG) layout algorithm is proposed to show the *job view* compactly and clearly. Then, novel anomaly scores are designed and visualized in the *performance matrix* to help users identify suspicious jobs and machines. Next, the execution status of the atomic tasks is shown in the *task view*, where a scatter-plot-based visualization allows users to observe the patterns of the tasks and identify the noteworthy tasks. We also incorporate a suite of auxiliary views to assist the users to reason about the discovered abnormal patterns. In particular, the *entity list* shows the statistics and detailed information of the components in query execution (e.g., job, task, and machine), and the *profiling view* is aligned with the *task view* to help users associate machine status with task execution. We implement well-coordinated linkage mechanisms for related items in different views and provide flexible interactions to navigate among the views, which makes interactive visual exploration easy.

QEVIS has been tested in the many real-world applications, which helps the engineers gain intuitive understandings of the query execution process and easily identify the problems in query execution. QEVIS is primarily developed for Apache Hive, however, its architecture and design elements (e.g., *job view* and *task view*) can be adapted to other systems, including Spark and Flink, that use Directed Acyclic Graph (DAG) as their computational abstraction.

In particular, we make the following contributions:

- **Multi-grained visualization framework for query execution.** We summarize comprehensive design requirements for the visual analysis of query execution and propose a visualization framework where multiple views complement each other by visualizing query execution at different granularities.
- **New layout algorithm to depict execution overview.** We design a novel algorithm to visualize the execution process of the jobs in the query plan and their dependencies. Compared with existing solutions, our approach optimizes the job layout to display the logic structure clearly while utilizing a smaller rendering space.
- **Novel scoring methods to identify execution anomalies.** We devise two scoring methods to evaluate the anomaly degree of the tasks spawned by a job. Different from existing scores that consider a specific anomaly, our scores generalize across different anomalies by measuring how far query execution deviates from the ideal case.
- **Scatter-plot-based visualization to show massive tasks.** We propose a scatter-plot-based visualization to display the execution details of massive atomic tasks. Compared with existing timeline-based visualizations, our visualization shows the distribution of the massive tasks intuitively and allows users to easily observe important normal/abnormal patterns in query execution.
- **Visual analytics system to coordinate multiple views.** We build QEVIS for Apache Hive, which properly coordinates the visualizations discussed above. Cross-view linkage and rich interactions are provided to support flexible navigation among the views.



```
SELECT sum(lo_revenue) AS lo_revenue, d_year, p_brand1
FROM lineorder, dates, part
WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey AND p_category = 'MFGR#12'
GROUP BY d_year, p_brand1;
```

(a) SQL query

(b) Logical plan

Resource scheduler
CPU RAM t_1^{M1} t_2^{M1} t_1^{R2} t_4^{R2} t_7^{R2} t_1^{R3} t_4^{R3} t_5^{R3} Machine 1
CPU RAM t_3^{M1} t_1^{M5} t_3^{M4} t_3^{R2} t_5^{R2} t_6^{R2} Machine 2
CPU RAM t_1^{M4} t_2^{M4} t_2^{R2} t_5^{R2} t_8^{R2} t_2^{R3} t_3^{R3} Machine 3

(c) Scheduling and executing tasks of each job

Fig. 2: The workflow of distributed query execution in Apache Hive

2 Background

To provide background for subsequent discussions, we take Apache Hive as an example to introduce distributed query execution process.

Figure 2 depicts the workflow of query execution in Apache Hive. When a query (e.g., the SQL-like statement in Figure 2(a)) is issued to the system, the query optimizer (e.g., Calcite [12]) and parallel execution engine (e.g., Tez [42]) optimize the query to a execution plan, which is the direct acyclic graph (DAG) of Map/Reducer jobs shown in Figure 2(b). Each Map or Reducer job in the DAG can be divided into three phases: (1) input, (2) processing, and (3) output. The input phase loads data from disk or upstream jobs. The processing phase executes a sequence of SQL operators, for example, the processing phase of Map 1 encompasses three SQL operators: table scan (TS), filter (FIL), and select (SEL). The output phase generates the output data, which can be the final result or input data for downstream jobs. The dependencies among the jobs are determined by their input/output relation and modeled by the edges in the DAG.

To run each Map/Reducer job, the execution engine (e.g., Tez) instantiates a set of tasks. All tasks of a specific Map/Reducer job run the same sequence of operators but work on different data partitions. For example, the execution engine instantiates three tasks, e.g., t_1^{M1}, t_2^{M1}, and t_3^{M1} (the gray cells in Figure 2(c)) for job Map 1 in Figure 2(b). Each task is scheduled to run on an executor (e.g., container in Apache Hive), which is a logical collection of physical resources (e.g., CPU and memory) on one machine in the cluster, by the resource scheduler (e.g., Yarn [51]). Task is the atomic unit for execution and scheduling, and the tasks of different jobs can run on the same machine. For example, as illustrated in Figure 2(c), both t_1^{M1} (a task of Map 1) and t_1^{R2} (a task of Reducer 2) are executed on Machine 1.

Referring to the workflow in Figure 2, many research work mainly focuses on visualizing query logic (Figure 2(a)) and optimization plans (Figure 2(b)). These studies are very helpful in finding bugs in query logic or optimization. QEVIS is orthogonal to them as it analyzes query execution traces (Figure 2(c)) in a distributed environment, and identifies the performance issues that might be caused by the scheduling mechanism, resources, or hardware, etc.

3 Related Work

In this section, we summarize the related work into two categories: query execution diagnose and visual understanding for query execution.

3.1 Query Execution Diagnose

Many systems [1, 4, 28, 35, 36, 48, 54] design dedicated scores and algorithms to diagnose problems in query execution. For example, Dr. Elephant [4] uses configurable rule-based heuristics to evaluate queries executed on Hadoop or Spark and gives suggestions (e.g., increasing virtual memory) to improve query performance. PerfXplain [30] offers a query language to express performance queries, along with a decision tree-based algorithm that generates explanations by analyzing a log of query executions. DBSherlock [54] allows users to select a time range that contains anomalies and analyzes the execution statistics and system configurations to explain the anomalies with a causality-based model. iSQUAD [36] requires users to label abnormal queries and then perform root cause identification using a Bayesian model. PerfDebug [48] is a specialized tool designed to analyze computation skew issues. Using data provenance-based techniques, it can automatically identify the records responsible for abnormalities. We refer interested readers to a comprehensive survey [23] on database query debugging. These systems work as a black-box that delivers the final diagnosis results (which may be inaccurate) without providing evidence. Moreover, their scores and algorithms are designed for specific anomalies, e.g., garbage collection time and data skew, but it is difficult to define a complete set of anomalies beforehand in practice and the causes of execution problems can be complex. Instead of considering a specific anomaly, the anomaly scores in QEVIS generalize across anomalies by measuring how far query execution deviates from the ideal case. Moreover, QEVIS provides multiple views and allows to identify the evidence and causes of execution anomalies via interactive exploration, which is more suitable for handling complex execution problems.

3.2 Visual Understanding for Query Execution

Understand query statement and logical plan. As explained in Section 2, a query is transformed into a logical plan before execution. In real-world applications, queries often stem from adjusting existing SQL templates and can be complex and challenging to comprehend. Prominent methods such as GraphSQL [16], Visual SQL [27], and QueryVis [31] employ node-link diagrams to visualize the topological structure of query sub-steps. Notably, GraphSQL [16] and Visual SQL [27] propose visual query languages to facilitate interactive query visualization and adjustment. QueryVis [31] and STRATISFIMAL LAYOUT [19] introduce visual formalisms that offer expressive visual encodings of SQL query meanings while minimizing redundant visual elements to enhance query logic clarity. Node-link diagrams are also commonly used to represent the logical structures of execution plans, which can be modeled as trees or DAGs [8, 9, 11, 39, 46]. Perfopticon [39], for instance, utilizes a nested node-link diagram to demonstrate the hierarchical structure of the logical plan for Myria [26] , a distributed big data system similar to Hive. Understanding query logic and execution plans is crucial for diagnosing query and optimization logic bugs, an aspect distinct from our primary focus. Understanding query logic and execution plans plays a critical role in diagnosing bugs associated with query and optimization logic. Nevertheless, our research supplements this area but primarily does not focus on it. QEVIS leverages dagre [3], a DAG layout library, to visualize query logic and optimization plans, enabling users to inspect them as required.

Understand query execution progress. The execution process of a query can be modeled as a sequence of job or task events with dependencies. We briefly discuss representative visualization methods for event sequence and refer interested readers to a comprehensive survey in [25]. *Gantt chart-based visualizations* are commonly used to show execution progress and widely adopted by execution trace visualization tools [8, 9, 11, 20, 33, 34, 39–41, 43, 53]. These works often organize the tasks executed by the same executor (e.g., CPU) into a row and utilize links to mark the relations between tasks [40, 41, 53]. Gantt chart-based visualizations are effective and intuitive when the number of tasks is small but they become cumbersome when there are many tasks [45], which is usually the case for distributed query execution. For instance, as relations are displayed as links, the crossing of many links can result in severe visual clutter, which makes the visualization difficult to read. Other *Chart-based visualizations* such as line-chart or scatter plot, are employed to show trends and distributions of events [18, 24, 37, 38, 45, 52]. Line charts are widely used to show the trend of performance metrics or aggregated metrics of resources measured during the execution process [22, 29, 32]. The scatter plot technique is commonly employed to provide a high-level overview of event sequences or event groups by projecting them onto a 2D canvas using dimension reduction techniques or selected attributes [24, 37, 45, 52]. In addition to depicting distributions, scatter plots are also utilized for outlier identification [18, 38]. One advantage of the scatter-based design is its scalability, allowing for effective visualization of large-scale event datasets. For instance, scatter plots have been effectively employed to identify important patterns in large-scale MPI (Message Passing Interface) events using various encoding strategies [45], which has inspired our *task view* design. Taking inspiration from existing approaches, QEVIS incorporates both Gantt-chart-based and scatter-based visualizations to accommodate the requirements of visualization at different scales. For the high-level visualization, where the scale is relatively small, QEVIS employs a Gantt-chart enhanced with links to intuitively display the execution of jobs and their dependencies. When dealing with a large number of tasks, QEVIS implements a scatter-based visualization design to effectively manage the complexity.

Interactive analytic tools. As discussed in Section 2, the query execution process is complex, and thus a fixed visualization is insufficient for understanding. DBSherlock [54] and iSQUAD [36] provide a visual interface that allows users to interactively select queries or time ranges with anomalies, and design algorithms to find the causes of the anomalies. Open-source tools such as TezUI [9] and SparkUI [8] allow users to observe the execution plan, execution progress, and other aspects of query execution in separate views. However, these tools lack cross-view linkage and flexible interactions, which makes it difficult for users to diagnose execution problems caused by complex reasons. VQA [46] is capable of monitoring and visualizing the real-time execution process of an input query. However, it is only limited to the single-machine environment. Perfopticon [39] is a work highly relevant to QEVIS, which utilizes four coordinated views to meet the complex analytical needs of analysts and a multi-granularity analysis approach to showcase the execution process of queries. The advantage of Perfopticon lies in its ability to associate operator execution with query execution, enabling analysts to examine the performance issues caused by operator logic. Additionally, Perfopticon allows users to view the execution status of a given fragment on each worker, quickly identifying which workers may be stragglers during fragment execution. Different from the worker-based analysis in Perfopticon, the analysis approach of QEVIS is task-based. Specifically, the execution engine instantiates a given job as a set of tasks which are then assigned to workers for execution. QEVIS visualizes patterns of a large number of tasks to reveal issues in the query execution process. Moreover, QEVIS takes into account the dependencies between jobs and tasks, enabling analysts to more accurately analyze performance issues related to dependencies, such as abnormal waits and deadlocks. Furthermore, QEVIS introduces modules such as anomaly scoring and system profiling, which help analysts conduct causality analysis more efficiently.

4 Design Tasks of QEVIS

The QEVIS project follows the guidelines of the design study methodology [44]. To start, we conducted semi-structured interviews with potential users of QEVIS from various backgrounds and collected the questions they commonly encounter in daily work. Then, we selected four domain experts as long-term collaborators, including two researchers (P1, P2) from academia and two engineers (P3, P4) from industry. P1 works on big data and database systems, while P2 focuses on query optimization. P3 is the leader of a team that focuses on cloud computing. P4 is a senior engineer working on distributed query system maintenance. Note that P1 is also an author of this paper. According to the feedback from potential users and domain experts, we formulate three major design requirements for QEVIS as follows:

R1. Understand query execution. Query execution can be understood at two scales: job scale and task scale. As an overview, the timing information of the jobs and their dependencies can provide a big picture of query execution. In this view, the analysts can know the general time usage of each high-level component and identify execution bottlenecks by analyzing the job dependencies. To gain a more accurate and comprehensive understanding of the query execution process, a micro-level view is necessary. This involves visualizing the execution progress and data dependencies of the atomic tasks. Such detailed visualization enables analysts to identify the required optimizations and explore potential solutions for addressing the bottlenecks.

R2. Identify the component worth paying attention to. Identifying key components is crucial for analysts to effectively navigate among the multiple levels of granularity. To facilitate this process, it is necessary to provide sufficient information that enables analysts to pinpoint the essential jobs, machines, and tasks that should be explored.

R3. Reason about specific execution patterns. After identifying the suspicious jobs or tasks, the next step is to find the reasons that cause their abnormal behavior. To accomplish this, analysts should analyze the execution of jobs and tasks, along with relevant information such as machine profiling. Moreover, all of the information should be appropriately visualized and coordinated to enable effective explorations.

Based on the requirements above, we formulate the design tasks as:

T1 Multi-grained visualization. To help analysts understand the query execution process (R1) and identify the key components (R2), it is necessary to display the execution process and key attributes at different granularities with effective visual designs:

T1.1 Show the overview of query execution. QEVIS should effectively visualize how the query logical plan is executed as an overview of query execution. Specifically, the timing information (e.g., start time, end time, and time usage) of each job should be shown intuitively for users to understand which components are time-consuming (**T1.1.1**). Other attributes such as the task parallelism and time usage percentage of each sub-phase of a job should be provided when users inspect a specific job (**T1.1.2**). Moreover, the dependencies of the jobs should be shown clearly for analysts to know the logic position of a specific job (**T1.1.3**).

T1.2 Support fine-grained task-level analysis. QEVIS should effectively visualize the timing information of many atomic tasks (**T1.2.1**) and their dependencies (**T1.2.2**). The different execution statuses should be revealed by distinctive visual patterns provided by the system. Moreover, the users should be able to identify the key tasks affecting the whole execution progress and drive the potential problems causing the performance issues.

T2 Anomaly score and visualization. To help users navigate among different analysis granularities and identify the components that are worth paying attention to (R2), general anomaly scoring methods should be integrated to find the jobs and machines that have abnormal behavior and effectively narrow down the scope of analysis.

T3 Enable pattern reasoning. To reason about the execution patterns (R3), auxiliary information (e.g., such as data and system profiling) should be provided on demand, and this information should be properly linked with the execution visualizations for interactive exploration.

T3.1 Provide rich auxiliary information to reason about execution anomalies. QEVIS should provide auxiliary information to help users understand query execution process and interpret anomalies. The distribution of task attributes, such as data input, data output and time usage, should be provided to reason about system behavior (**T3.1.1**). Moreover, machine profiling (e.g., memory and CPU usage) should be visualized jointly with the tasks such that users can correlate task execution with machine status (**T3.1.2**).

T3.2 Provide coordinated linkage among different views. When inspecting a specific pattern or abnormal component, users need to switch among different views. Related components (e.g., job, task and machine) in different views should be well linked to make interactive exploration easy.

5 The QEVIS System

The architecture of QEVIS is illustrated in Figure 3. In this section, we first introduce the system architecture in Section 5.1. Then we discuss data model of QEVIS in Section 5.2 and elaborate on the visualization designs in Section 5.3.

5.1 System Architecture

We developed a prototype of QEVIS and kept improving it according to user feedback. Figure 3 illustrates the final system architecture and analysis pipeline of QEVIS. The system consists of three modules for (1) data collection, (2) data analysis, and (3) visualization and interaction. When users run a query, the data collection module collects and processes incoming query execution logs with the stream log engine. Specifically, it persists the basic information and passes logs to the database until execution status indicators (i.e., *Pause*, *Terminate*, *Complete*, *Timeout*) are detected. To monitor query execution at run time, the stream log engine redirects processed logs to the frontend for visualization. In the meantime, the logs are stored in a database to enable in-depth analysis of completed queries. The data analysis module reads data from the database and calculates anomaly scores. The visualization and interaction module displays the query execution process and provides rich interactions to support interactive exploration.

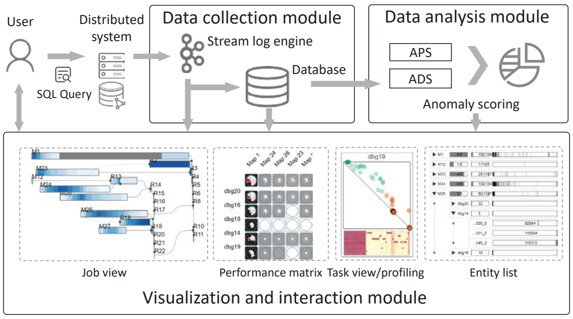


Fig. 3: QEVIS system includes three modules: (1) data collection module, (2) data analysis module, and (3) visualization and interaction module

5.2 Data Model

As introduced in Section 2, a query is converted to a DAG of Map/Reducer jobs during query execution. We denote the DAG as $G(J,R)$, where J is the node set, and each node is a Map or Reducer job in the logic execution plan; R is the edge set, which models the dependencies among the jobs. For each atomic task, we divide its execution process into three phases : (1) input, (2) processing, and (3) output, for fine-grained analysis. We use $t_{exe} = \{(t_s^i, t_e^i), (t_s^p, t_e^p), (t_s^o, t_e^o)\}$ to denote the start time and end time of each phase in task t. Thus, the data model of task t is a tuple $\langle t_s, t_e, t_j, t_m, t_{exe} \rangle$, which are the task's start time, end time, job identifier, machine identifier and the execution phases, respectively. The data model of each job is a tuple $\langle j_s, j_e \rangle$, which are the job's start time and end time, respectively. A job has many tasks, for job x, its start time j_s is defined as the earliest start time of all its tasks, i.e., $j_s = \min\{t_s | t_j = x\}$, and its end time j_e is the latest end time of all its tasks, i.e., $j_e = \max\{t_e | t_j = x\}$. With the data model above, the original job DAG is augmented to a temporal DAG (TDAG), which contains start/end time for each job and is key to query execution analysis.

5.3 Visualization Designs in QEVIS

Figure 1 illustrates the user interface of QEVIS. With a selected query from the *query list* (Figure 1(a)), *job view* (Figure 1(b)) appears to provide the overview of query execution (**T1.1**). We devise a novel algorithm to show the timing information of jobs and their dependencies, which is elaborated in Section 5.3.1. In Section 5.3.2, we provide a *performance matrix* (Figure 1(c)) to show the anomaly degree of the jobs and machines. *Performance matrix* helps users quickly narrow down to the jobs and machines worth paying attention to (**T2**). In Section 5.3.3, we design the *task view* (Figure 1(d)) to support fine-grained task analysis (**T1.2**). In Section 5.3.4, we present the auxiliary views for detailed information (**T3.1**) and cross-view linkages (**T3.2**).

5.3.1 Job view

To effectively visualize the progress of job execution, it is essential to display the timing information, including start time, end time, and time usage, as well as the dependencies among jobs in the query logical plan (**T1.1.1** and **T1.1.3**). This data can be modeled as a DAG enhanced with temporal information (TDAG). Existing tools (e.g., TezUI and SparkUI) enhance traditional Gantt chart with links to visualize the TDAG. This solution is limited because (i) it does not consider the visual clutter caused by the many job dependencies in the execution

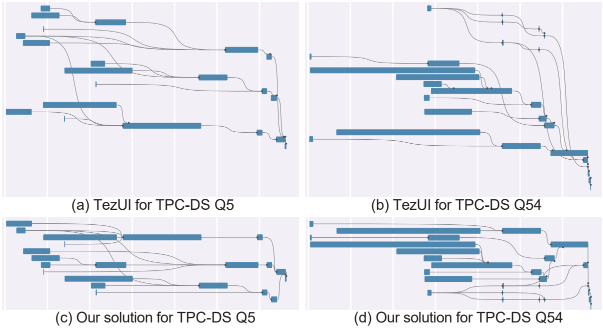
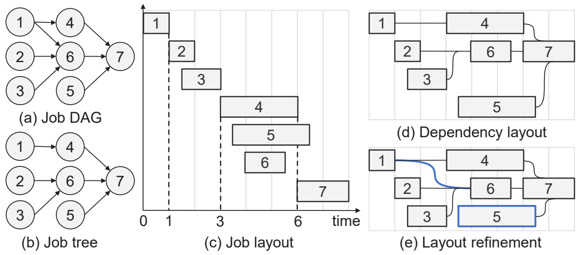

(a) TezUI for TPC-DS Q5
(b) TezUI for TPC-DS Q54
(c) Our solution for TPC-DS Q5
(d) Our solution for TPC-DS Q54

Fig. 4: Comparing TDAG layout methods



(a) Job DAG
(b) Job tree
(c) Job layout
(d) Dependency layout
(e) Layout refinement

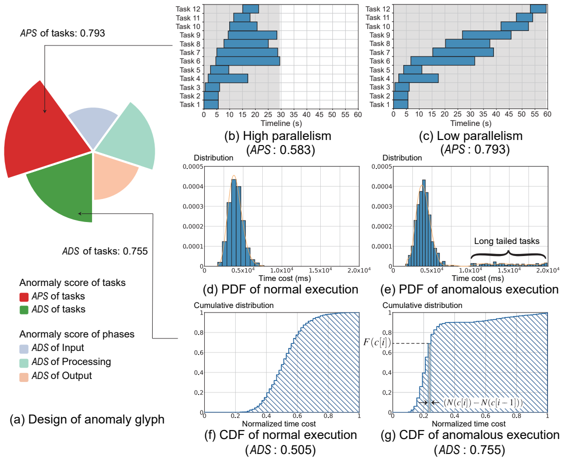
Fig. 5: Workflow of our TDAG layout algorithm

plan; and (ii) it does not scale to large TDAG as screen space is not utilized efficiently. Figure 4 (a) and Figure 4(b) show its visualizations for the TDAG of TPC-DS query 5 and 54, respectively. The edges and nodes are crossed in Figure 4(a) due to complex job dependencies, and there is a large blank space in the left bottom part of Figure 4(b).

To tackle the two challenges, we propose a novel TDAG layout method illustrated in Figure 5. It includes the following steps:

- **Step 1.** We first simplify the job DAG, see Figure 5(a), to a job dependency tree in Figure 5(b). In particular, the output of a job can be the input of many jobs (e.g., ≥ 2 jobs), and we only preserve the edge from a job to its out-neighbor job with the earliest start time. For example, the output of job 1 serves as input for jobs 4 and 6 but the start time of job 4 is earlier than job 6. Thus, we only keep the edge from job 1 to job 4 in the simplified job tree in Figure 5(b). TDAG simplification is conducted to (i) preserve the starting time order of the jobs, and (ii) keep dependent jobs close to each other to prevent visual clutter.
- **Step 2.** We then plot a rectangle for each job by using the length of the rectangle to indicate job duration and sort the jobs by their start time, as shown in Figure 5(c). This follows human reading habit (i.e., reading from top to bottom) and plots the job starting earlier in a higher position.
- **Step 3.** We next adjust the layout of jobs by utilizing the job dependencies in the simplified tree. In particular, we check the jobs the top to bottom. For each job, if its out-neighbor job could be placed in the same row with it (i.e., non-overlapping), we move the out-neighbor to the same row with it, and add an edge between these two jobs. For instance, the time duration of job 1 does not overlap with the time duration of its out-neighbor job 4, thus we plot jobs 1 and 4 in the first row, see Figure 5(c).
- **Step 4.** Last, we refine TDAG layout by (i) adding the other edges in the job DAG, for example, jobs 1 and 6 in Figure 5(e), and (ii) reducing the space by combining the rows that do not overlap, for example, job 3 and job 5 are plotted in the same row in Figure 5(e).

We visualize the TDAG of TPC-DS query 5 and query 54 by our TDAG layout algorithm in Figure 4(c) and Figure 4(d), respectively. When compared with the visualizations produced by Tez UI Figure 4(a) and Figure 4(b), the visualization generated by our approach offers two advantages: (1) our visualization takes into account the topological structure of the execution plan, resulting in reduced visual clutter as there are fewer crossings among the links and rectangles; (2) our proposed visualization optimizes screen space utilization by considering both dependencies and time duration, allowing for a more efficient visualization.



(a) Design of anomaly glyph
(b) High parallelism ($APS: 0.583$)
(c) Low parallelism ($APS: 0.793$)
(d) PDF of normal execution
(e) PDF of anomalous execution
(f) CDF of normal execution ($ADS: 0.505$)
(g) CDF of anomalous execution ($ADS: 0.755$)

Fig. 6: Illustration of anomaly scores

We enhance the *job view* with two visual designs (i.e., parallelism and phase mode) to show more information for each job (**T1.1.2**).

Parallelism mode: we embed the task parallelism (i.e., number of running tasks) into the job rectangle. Specifically, we use a 1D-heatmap to visualize the number of active tasks over time, as shown in Figure 1(b). A gradient color from white to dark blue encodes the number from one to the maximum number of active tasks for the job. We color the number zero with a special color (e.g., gray) as zero indicates that the execution of the job is suspended and users should pay attention.

Phase mode: we visualize the percentage of time used by the three phases of each job, as shown in Figure 1(b1), and use three colors to indicate the three phases. This allows users to better understand the jobs, e.g., classifying them into I/O-bounded or compute-bounded jobs.

5.3.2 Performance matrix

As shown in Figure 1(c), performance matrix is designed to show the anomaly degree of jobs and machines such that users can narrow down the scope of exploration (**T2**). Existing tools [4, 6] propose different scoring schemes for different anomaly types, e.g., data skew and memory overload. This approach is limited as it is difficult to encompass all possible anomalies. We adopt a different approach, which considers the execution statistics of the tasks and quantifies how far they deviate from ideal distributed parallel execution. We believe a component is worth paying attention to if it deviates far from ideal, and we leave it to the users to determine the specific anomalies by inspecting the views we provided. In the ideal case, all tasks of a job have a similar workload and run in parallel, which results in similar time usage and high parallelism for these tasks. On this basis, we propose two novel anomaly scores to analyze the tasks of a job, i.e., *abnormal parallelism score* (APS) and *abnormal duration score* (ADS).

Abnormal parallelism score (APS). APS measures how well the tasks of a job are parallelized, how many jobs share the similar start time and end time.

For example, the tasks in Figure 6(b) are paralleled better than the tasks in Figure 6(c) as the tasks are better overlapped in time and more of them run concurrently. Denote the task set of job j as $T_j = \{t_j[1], \cdots, t_j[n]\}$, the APS of T_j is defined as:

$$\mathrm{APS}(T_j) = 1 - \frac{\sum_{i=1}^{n}(t_j[i]_e - t_j[i]_s)}{n \times (j_e - j_s)}.$$

Intuitively, the second term of APS measures the ratio between the area of all tasks (the bars in Figure 6(b) and Figure 6(c)) over the area of the job rectangle (the gray area in Figure 6(b) and Figure 6(c)). If all tasks have the same time usage and run concurrently, APS will be 0, and thus large APS indicates anomalous. For instance, the APS for the tasks in Figure 6(b) and Figure 6(c) are 0.583 and 0.793, respectively, indicating that Figure 6(c) deviates further from the ideal.

Abnormal duration score (ADS). It is observed that the time usage of the tasks in a job is usually tightly clustered and unexpectedly long query execution time is usually caused by a few long-running tasks [47], which causes a long tail in the time usage distribution. For instance, Figure 6(d) and Figure 6(e) illustrate the task time usage distribution of two jobs, and the distribution in Figure 6(e) has a long tail, which results in a long job time usage.

We use ADS to measure the significance of long tail tasks. Denote the task set of job j as $T_j = \{t_j[1], \cdots, t_j[n]\}$ and the time usage of the tasks as $C_j = \{c[1], c[2], \cdots, c[m]\}$ when sorted in ascending order. The ADS of T_j is defined as follows:

$$\mathsf{ADS}(T_j) = \sum_{i=2}^{m} F(c[i]) \cdot (N(c[i]) - N(c[i-1]))$$

where $F(c[i])$ is the cumulative distribution function of task time usage, which returns the percentage taken by the tasks with shorter time usage than $c[i]$ in all tasks, and $N(c[i]) = \frac{c[i]-c[1]}{c[m]-c[1]}$ is the normalized value of $c[i]$ and is in the range of 0 to 1 for all tasks. Intuitively, ADS measures the area covered under the cumulative distribution of task time usage, the gray areas in Figures 6(f) and (g), and long tail tasks will enlarge the area and ADS. The intuition of the ADS score is measuring the cumulative distribution of all tasks. The ADS score will be large if some tasks have unexpectedly long running time. For instance, the ADS of the tasks in Figures 6(d) and (e) are 0.505 and 0.755, respectively, indicating that Figure 6(e) has a more severe long tail problem.

We scale the APS and ADS scores of a job j by the time usage of the job in the entire query, i.e., $\rho(j) = \frac{j_e - j_s}{C}$, where C is the overall execution time of the query. This allows users to focus on important jobs because even if a short-running job has execution anomalies, its influence on the entire query may still be small. Our two anomaly degree scores are generic and can assist analysis at different levels, e.g., job level, task level or phase level. As will be shown later, we also use them to measure the anomaly degree of the three phases in the tasks.

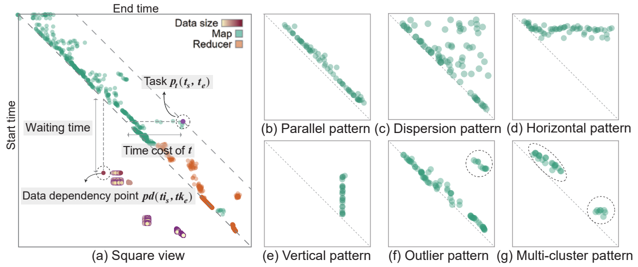
Visualization designs for performance matrix. As shown in Figure 1(c), we use a matrix-based visualization to show the anomaly scores. The x-axis and y-axis correspond to the jobs and machines, respectively. The first row displays the overall anomaly scores of each job. The cell in m-th row and j-th column displays the anomaly score of job j's tasks that are executed on machine m. We sort the columns and rows by descending order of their average anomaly score. In cases where the number of machines or jobs is particularly large, we preserve only the top k jobs (default value: 30) and top n machines (default value: 10) as specified by the analyst. As illustrated in Figure 6(a), we show the anomaly scores by Nightingale's chart [14], which is commonly used to visualize multi-attribute data [55]. The radius of each sector represents the value of an anomaly score while the color represents the granularity (e.g., task- or phase- level) of anomaly. The color encoding for the phases is coincident with the job progress in Figure 1(b1).

5.3.3 Task view

The *task view* allows users to analyze the tasks in detail (**T1.1.3**) and consists of multiple panels . The panel at the top of *task view* shows all tasks of the query, and the remaining panels display the tasks executed on each machine. Each panel displays the machine id (e.g., dbg19) at the top and uses a scatter plot-based visualization to plot the tasks as points in a square view, as shown in Figure 7(a).

The visualization provides two kinds of task information using different parts: (i) *Temporal distribution*, which depicts the start time, end time and time usage of the tasks, see the top-right triangle in Figure 7(a); and (ii) *Data dependency distribution*, which embeds the data dependencies among the tasks of different jobs, as shown by the bottom-left triangle in Figure 7(a).

Temporal distribution. Existing tools (e.g., SparkUI [8] and Inviso [5]) usually employ Gantt chart to visualize the tasks, which are limited for two reasons: (i) Gantt chart causes severe visual clutter when visualizing massive (e.g., thousands) tasks; (ii) it is difficult to compare the time usage of different tasks as Gantt chart lacks proper alignment.



(a) Square view (b) Parallel pattern (c) Dispersion pattern (d) Horizontal pattern (e) Vertical pattern (f) Outlier pattern (g) Multi-cluster pattern

Fig. 7: *Task view* and representative patterns

To tackle these limitations, we plot the tasks as points in a square view. In particular, the x-axis denotes the task start time and the y-axis the end time, turning task t into a point $p_t(t_s, t_e)$. This configuration of the axes aligns with the conventional reading patterns observed in humans when browsing websites. We use green and orange colors to encode the tasks of map and reducer job, respectively. As shown in Figure 7(a), this scatter point view design has several properties: first, all points are in the top-right half of the square view because $t_e > t_s$ holds for all tasks; second, the horizontal distance between the diagonal line and point p_t encodes the time usage of task t, as illustrated by the dashed line in Figure 7(a). When a user moves the mouse on a task t, we plot a gray dashed auxiliary line that is parallel to the diagonal line and intersects point p_t to show the time usage of the task. More importantly, for all tasks of a job, the distribution patterns of the points in the scatter point view reveal the query execution status and provide useful hints for the root causes of execution problems. Figures 7(b) to (g) show the most representative patterns we observed in the production environment. We provide our observed hints of them below.

- **Parallel pattern in Figure 7(b):** it suggests that all tasks have similar time usage, resulting in points that are parallel and close to the diagonal line. A dense and parallel pattern indicates the query is executed smoothly.
- **Dispersion pattern in Figure 7(c):** it means that the tasks of a job have different start time and end time. It suggests that these tasks may encounter multiple problems at the same time, such as insufficient resources, load imbalances, and limited executors, etc.
- **Horizontal pattern in Figure 7(d):** it means all tasks start at almost the same time but end at different times. As a special case of Dispersion pattern, it indicates the number of containers is enough, since all tasks can be executed at the same time.
- **Vertical pattern in Figure 7(e):** it shows that the tasks finish together. The vertical pattern appears typically because these tasks are all waiting for input data from a same upstream task which is usually the bottleneck.
- **Outlier pattern in Figure 7(f):** it reveals that several tasks have much longer time usage than others. It indicates that these tasks may suffer from data skewness or deadlock. These tasks are largely the bottleneck of the query execution.
- **Multi-cluster pattern in Figure 7(g):** it shows several separated groups of tasks. It usually indicates the execution of the job is interrupted for a period of time, which is usually caused by hardware problems or inefficient resources.

Alternative design. Several design alternatives for *distribution view* are considered, including the Gantt chart, Marey's chart and Arc chart, which have been used in related research work and software. Figure 8 shows the implementation of these designs and our methods with the same dataset. The green and orange colors indicate Map and Reducer tasks, respectively. Gantt chart is the most intuitive design to visualize the tasks through rectangular bars, however, when data is large, the height of bars will become too narrow to be observed. The other two designs all use lines (curves) instead of bars to show the temporal information of tasks. Marey's chart uses two horizontal axes to indicate the start time and end time, thus the task can be represented as a line connecting the two axis. Arc chart uses arcs to show the tasks, with two end points of the arc indicating the start time and end time. All these designs work well when the data size is small. However, in our

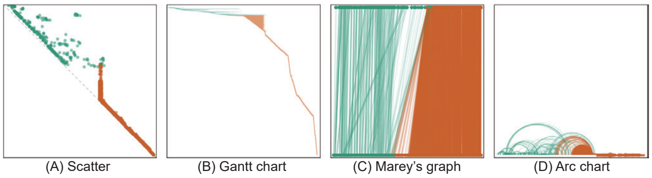


Fig. 8: Alternative design of task distribution

application, the number of tasks will reach tens of thousands, which results in different problems for each design. For instance, the height of bars becomes too narrow to observe in Gantt chart; Marey's chart and Arc chart all have serious visual clutter results from the overlap and cross of lines (curves), especially for Marey's graph, the orange lines are too dense to reflect any patterns.

Data dependency distribution. Consider two tasks ti and tk with data dependency, and the output of ti is used as input by tk. The start time of tk should be later than the end time of ti in the idle case but the opposite may happen in abnormal cases (usually caused by the resource scheduler). To visualize these abnormal cases, we show the data dependency of ti and tk by plotting point $p_d(tk_s, ti_e)$ in the square view. As $tk_s < ti_e$, all these abnormal tasks are in the bottom-left half of the square view, as shown by the purple points in Figure 7(a). Thus, the dependency distribution does not interfere with the temporal distribution, which is in the top-right half. We provide several interaction mechanisms in task distribution. For example, once a task is selected, its dependencies, upstream tasks and downstream tasks will be highlighted in red and blue colors, respectively.

5.3.4 Auxiliary views and interaction designs

We provide a suite of auxiliary views and rich cross-view interactions in QEVIS to help users explore query execution progress. Due to space limitation, we only present two auxiliary views (i.e., *entity list* and *profiling view*) and refer the interested readers to our GitHub repository for the other views (e.g., *query list* in Figure 1(a)).

Entity list: it shows detailed statistics of each job row by row, shown in Figure 1(e). Users can click on the triangle at the leftmost side of each job (i.e., job level) to enter the machine level. By clicking the triangle at the machine level, all tasks executed by the machine will be listed. To visualize the features of tasks, we implement two visualization forms, i.e., rug plot [7] and Gantt chart, that will be selected automatically based on the features of interest. Besides time usage, *entity list* also allows users to select other features of interest (e.g., read/write data size) through the dropdown at the top of this view.

Profiling view: it is embedded into each panel in the *task view* and consists of a suite of visualizations to show execution statistics such as task parallelism and system hardware usages as illustrated in Figure 1(f). For single variate features such as task parallelism and memory usage, we use the line chart, which can show the temporal change of the value. For multiple variate features such as CPU usage and Disk IO, we use a heatmap, which is widely used to visualize multiple values.

The design of interactions. QEVIS supports flexible interactions and cross-view linkage to facilitate multi-view exploration. For example, when the user hovers the mouse on a job in the *job view*, shown as the purple boundary in Figure 1(b), all tasks of this job will be highlighted in the *task view*, as purple points in Figure 1(d). Conversely, if the user hovers the mouse on a task point in the *task view*, the corresponding job in the *job view* and *entity list* will be highlighted. Moreover, when the user clicks on points in the *task view*, the *entity list* will be expanded to show the corresponding tasks and execution machines, shown as the purple items in Figure 1(e). When the user puts the mouse on an element for more than three seconds, a widget showing detailed information about the element will be displayed. Shown as Figure 1(d2), when we put the mouse on the purple task, the task information such as data read, records processed, and time costs of each phase are displayed.

6 Empirical Evaluation

In this section, we show how QEVIS helps domain users understand query execution and pinpoint problems with three real cases from the production environment. Subsequently, we invite engineers from our industry partner to use our system and collect their feedback.

6.1 Case study

QEVIS has been widely used by software engineers for various real-world applications. Usually, our users are interested in queries that run longer than expected because they degrade system throughput. In this section, we demonstrate the effectiveness of QEVIS via three use cases about slow queries in production, which identifies hardware, system and data problems during query execution, respectively.

6.1.1 Identifying hardware problem

Figure 1 shows how QEVIS visualizes a long query (i.e., 623 seconds) from a business intelligence application and we analyze it as follows.

Step 1: investigate the performance bottleneck via *job view*. As shown in Figure 1(b), job M1 has a very long running time. More importantly, there is a large gray region in job M1, which means that the number of running tasks is 0 in this region according to the design in Section 5.3.1. Thus, job M1 is the cause of long execution.

Step 2: inspect job M1's task execution pattern via *task view*. To find the reasons that cause the gray region for job M1 in the *job view*, we analyze the tasks of job M1 via the *task view*. When hovering the mouse on M1, its associated tasks are highlighted with purple color in the *task view*, see Figure 1(d). We observe that the tasks of job M1 form two groups (i.e., **multi-cluster** pattern) that are far away from each other, as illustrated by the two dashed ellipses in Figure 1(d). The multi-cluster pattern suggests that the tasks of job M1 are not executed properly due to machine problems (discussed in Section 5.3.3).

Step 3: locate the problematic machine via *entity list*. We know that the long running query is caused by machine problems but it is difficult to pinpoint the problematic machines as the production cluster is large. Fortunately, QEVIS provides the *entity list* to map tasks to their executing machines. By checking the *entity list*, as illustrated by the rows with purple stroke in Figure 1(e1), we observe that the tasks of job M1 in the right-bottom corner are executed on machine dbg18 Then, we take a closer look at the tasks executed on dbg18, see Figure 1(d1). Compared with the other machines (e.g., dbg16, dbg19, dbg20), the number of tasks executed by dbg18 is quite small, which suggests that the resource scheduler YARN allocates a small number of tasks to dbg18. This confirms that dbg18 is the problematic machine and YARN is aware of this fact during the query execution progress.

Similar analyzing steps are also applied to job M24, which is another long running job in the query. We find that the long running time of job M24 is caused by a few straggling tasks on dbg19. By investigating them, we find that dbg19 has high CPU utilization, see the red regions of the CPU usage heatmap in Figure 1(f). We omit its analyzed steps as they are similar to the steps for job M1.

To sum up, hardware problems cause the long running query. To verify, we remove dbg18 and dbg19, and re-run the query. The running time becomes 200 seconds. Figure 1(b1) shows the new *job view*, where the running time of jobs M1 and M24 are much shorter than before.

Discussion. In this case, the query execution suffers from multiple hardware issues. The *task view* of all tasks is crucial in quickly identifying abnormal tasks and the *task view* of dbg18 shows dbg18 is suffering from a hardware problem. This scenario demonstrates that the visual pattern of tasks is helpful in diagnosing these issues efficiently.

6.1.2 Identifying system problem

We analyze a query that generates app downloads report for market analysis in this case.

Step 1: observe abnormal jobs via *performance matrix*. As shown by the *performance matrix* in Figure 9(a), the anomaly scores of jobs R3 and M8 are significantly larger than the other jobs, especially the abnormal duration scores (i.e., red sectors). Moreover, the green sector (i.e., abnormal parallelism score) of job R3 is also very large. Since job R3 is the downstream job of M8 as shown in the *job view* in Figure 9(b),

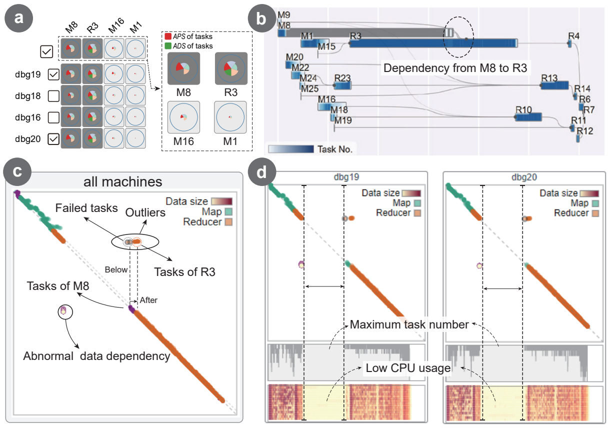
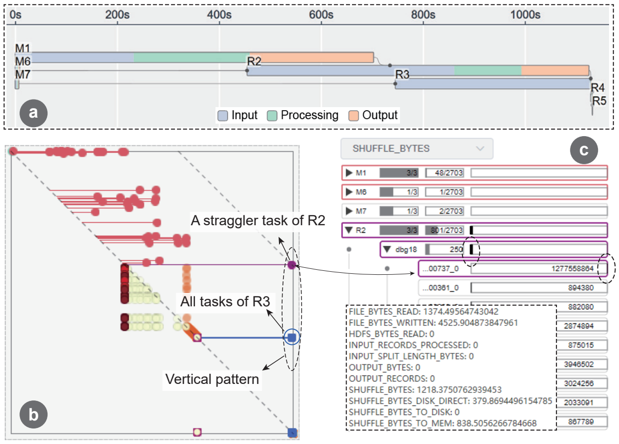


Fig. 9: QEVIS identifies system problem (task deadlock)



FILE_BYTES_READ: 1374.49564743042
FILE_BYTES_WRITTEN: 4525.904873847961
HDFS_BYTES_READ: 0
INPUT_RECORDS_PROCESSED: 0
INPUT_SPLIT_LENGTH_BYTES: 0
OUTPUT_BYTES: 0
OUTPUT_RECORDS: 0
SHUFFLE_BYTES: 1218.3750762939453
SHUFFLE_BYTES_DISK_DIRECT: 379.8694496154785
SHUFFLE_BYTES_TO_DISK: 0
SHUFFLE_BYTES_TO_MEM: 838.5056266784668

Fig. 10: QEVIS detects data skew

we conjecture that there is a causal relation between the abnormal behaviors of jobs M8 and R3.

Step 2: reveal abnormal data dependency via *task view*. We next perform fine-grained exploration on the tasks via the *task view*. There are several **outliers**, as highlighted in Figure 9(c). These outlier tasks can be classified into two categories: (i) failed tasks (the gray points), and (ii) long running tasks (highlighted in orange color). Interestingly, several tasks of map job M8 are executed immediately after these failed tasks, and then the orange colored tasks of reducer job R3 are executed. Visually, we can see there are several purple points (i.e., the tasks of job M8) vertically located below the gray points (i.e., failed tasks) and the orange points (i.e., tasks of job R3) are horizontally located after the purple points. Thus, we confirm that there are abnormal data dependencies among the tasks of jobs M8 and R3, shown as the data dependency points in the left-bottom part of Figure 9(c).

Step 3: reason about the failed tasks via *profiling view*. As the *profiling view* of the machine is aligned with *task view* by time, we observe that both dbg19 and dbg20 run the maximum number of tasks during the execution period of these failed tasks, see the range between the two vertical dashed lines for dbg19 and dbg20 in Figure 9(d). However, the CPU utilization of two machines is very low in this period. This suggests that the containers of the two machines are fully occupied but no work is done, and thus there is a task deadlock. In particular, the scheduler assigns all containers of dbg19 and dbg20 to the tasks of R3, and these tasks are waiting for the input data from the upstream tasks of job M8. But the upstream tasks cannot be executed as there are no idle containers, and thus the tasks of M8 and R3 form a circular waiting. The deadlock is resolved after killing several tasks of R3, as depicted by the gray colored failed tasks in Figure 9(c).

Discussion. Deadlocks occur when there is a suboptimal task scheduling. QEVIS addresses this by offering following key features: task distribution for quick identification of critical tasks, dependency visualization for understanding logical task relationships and alignment of execution patterns with system profiling to show the interplay between resources and execution. The conventional approaches may struggle to manage this situation due to their lack of task oriented visualization (both task and dependency) and task-system profiling alignment.

6.1.3 Identifying data problem

We next analyze a query that runs daily for sales analysis, which takes 1200 seconds and is longer than before.

Step 1: identify performance bottleneck via *job view*. We inspect its *job view* in Figure 10(a) and observe that the input phase dominates the running time of job R3. To find the reason for the long input time in job R3, we examine all its tasks via *task view* in Figure 10(b).

Step 2: locate the straggler task via *task view*. We find that (i) all tasks of job R3 are concentrated in a small region, and (ii) there is a straggler task of job R2, which are highlighted in blue and purple circles in Figure 10(b), respectively. Moreover, there is a **vertical** execution pattern among the tasks of job R3 and the straggler task of job R2, see the dashed eclipse, which suggests that the downstream tasks (of job R3) are waiting for the upstream task (of job R2).

Step 3: reason about the straggler task via *entity list*. By clicking the straggler task of job R2 in the *task view*, the row of this task is automatically located and highlighted with a purple boundary in Figure 10(c). We analyze its long running time by checking its processed data size, i.e., switching to "SHUFFLE_BYTES" in the dropdown menu. Surprisingly, the processed data volume of this straggler task is almost 1000 times of the other tasks in job R2, which indicates a severe data skew among the tasks. Thus, the above unusually long running query is caused by data skew problem.

Discussion. This case study highlights a scenario where a single task acts as the primary bottleneck in the execution process. Although multiple tasks exhibit abnormal execution behavior, correctly pinpointing this specific task is vital. This requires a visualization of tasks along with their dependencies, a feature uniquely offered by QEVIS.

6.2 User Interview

In this section, we discuss the feedback from the users of QEVIS.

6.2.1 Feedback from long term collaborators

The first set of feedback comes from our two long-term collaborators (i.e., P1 and P2), who helped formulate the design requirements and tasks of QEVIS in Section 4. The two experts collaborated with us for more than one year to build and improve QEVIS. We conducted semi-structured interviews with them face to face and asked them to evaluate QEVIS according to the design tasks.

Multi-scale visualization. Both P1 and P2 agree that the multi-scale visualization of QEVIS helps analysts conduct a wide range of query execution analysis tasks. P2 comments that the design of the *job view* is effective for the overview of query execution. He said "the *job view* is like a portrait and enables me to quickly comprehend the overall execution process of a query, which is particularly important when the same query is executed repetitively." Both P1 and P2 are satisfied with the scatter plot-based design of the *task view* as it shows representative distribution patterns. P1 said that "it provides a new visual form for analysts to quickly evaluate query execution and select the tasks of interest". However, both P1 and P2 expect more flexible interactions such as selecting a group of tasks by drawing a polygon and visualizing the common feature of the selected tasks.

Anomaly scoring and visualization. Both P1 and P2 believe that our general scoring methods are effective in measuring the anomaly degree. P1 suggests that the *performance matrix* complements the *job view* when the *job view* does not provide clear clue for the component to inspect. P2 claims that he always starts his exploration from the *performance matrix*, which also provides an overview of query execution. In addition, P2 recommends developing an anomaly score for the entire query, which provides a high-level summary of query execution and facilitates comparison between repetitive executions.

Enable pattern reasoning. P1 and P2 think that the design of the *entity* *list* is effective, especially when using multi-grained visualizations to explain the tasks with abnormal timing patterns. They also said that visually correlating system profiling with task execution helps explain abnormal task execution. However, P1 commented that the *task view* and *profiling view* are too small to inspect. He suggested allowing users to expand these views to a large separate display when necessary.

6.2.2 Feedback from potential users

Settings. The second set of interviews was conducted with six engineers (E1-E6) in a company to evaluate the usability of QEVIS. None of these participants used QEVIS before the study. We provided them with 10 real-world application cases. These cases exhibited various issues such as insufficient resources, incorrect configuration, and hardware problems. Participants were instructed to freely explore these cases with QEVIS and draw conclusions regarding the factors contributing to the slow query executions. The study was conducted on a screen with a resolution of 1980 × 1080, and each session lasted 80 minutes, including a 5-minute demonstration video, a 5-minute introduction to the study, a 15-minute introduction to QEVIS, and 55 minutes for free exploration. During the study, the participants were encouraged to think aloud, allowing us to collect feedback in real time. Then, they were asked to fill out a questionnaire with seven questions:

Q1 Which view is the most important during your exploration?
Q2 Which view do you use the most during your exploration?
Q3 How does QEVIS compare to similar tools you have used?
Q4 How easy is it to learn to use QEVIS?
Q5 How easy is it to navigate and find what you need?
Q6 Do you have any other comments or suggestions?

For **Q4** and **Q5**, the participants can rate the difficulty level on a scale from 1 to 7, with 1 indicating "very difficult" and 7 indicating "very easy". We also encouraged the participants to write down the reasons for their answers.

Feedback discussion. Regarding **Q1**, 4 out of the 6 participants identified the *task view* as the most important during their exploration. They noted that they needed to use this view to find the root causes of query execution anomalies. E1 selected the *job view* and said that he could inspect query execution by switching among different modes. E6 selected the *performance matrix* because it indicates the problematic machines.

In response to **Q2**, 4 out of the 6 participants also selected the *task view* as the most used view because they spent a lot of time switching between the *task view* and *entity list* to find the causes of long-running tasks. The *job view* and *entity list* were also selected by one participant.

Regarding **Q3**, only E1 has not used other query visualization tools before. E2-5 were familiar with TezUI, while E2 and E6 had experience with Dr. Elephant. E2 and E3 commented that QEVIS was more flexible than TezUI due to its interactions. "It can help me to quickly identify the execution pattern of the tasks in a job", claimed by E2. E3 stated that the *job view* helped him quickly compare different executions of the same query and he did not need to switch between different web pages. E6 thought that QEVIS was harder to use than the automatic query diagnosis in Dr. Elephant but agreed that QEVIS is more powerful in analyzing complex execution problems. He suggested integrating more algorithms to measure the anomaly of machines based on profiling and correlate machine anomalies with task execution.

In response to **Q4**, the participants gave an average score of 5.6, with a minimum score of 3 and a maximum score of 7. Participants that have used other visualization tools tended to be more positive about QEVIS (E2-5). However, E6, who gave the minimum score, commented that there were too many visual channels (e.g., color encoding, size, shape) in QEVIS and it is hard to remember them. He suggested adding more detailed legends to explain the meaning of the visual encodings.

Regarding navigation (**Q5**), the participants gave an average score of 6, ranging from 5 to 7. The participants stated that the cross-view linkage made it easy to find the elements of interest.

In summary, the user study suggests that QEVIS is a powerful and user-friendly tool for diagnosing runtime anomalies in distributed query execution, although there is still room for improvement.

7 Discussions and lessons learned

In this section, we discuss the lessons learned from the QEVIS project.

Collaboration. Effective collaboration with domain experts is crucial for the designs of the domain-specialized visualizations in QEVIS. By closely observing their problem-solving in real-world contexts, we can better discern and distill design requirements. It is beneficial for visualization researchers to immerse themselves in the domain experts' tasks. Similarly, domain experts should actively provide feedback on visual designs. For instance, when deciding on coordinate encoding in the *dependency distribution*, domain experts recommended displaying abnormal dependencies at the canvas's bottom left, ensuring efficient space utilization and highlighting critical information.

Overview vs. detail view. Choosing the right analysis granularity is important for solving specific tasks. Logical plan visualization (*job view*) is easy to understand and can help solve high-level problems. Atomic task visualization (*task view*), on the other hand, contains many tasks that are executed distributedly and is more difficult to understand. However, it is usually necessary for solving low-level problems such as query debugging and execution bottleneck identification. We recommend that the users choose different visualization granularities on demand. For instance, novice users can start with logical level visualization for a general understanding of the query execution process and narrow their exploration progressively, while experienced users can directly begin with the *task view* for in-depth analysis.

Generalization. While QEVIS was primarily developed for Apache Hive, its core components can be broadly applied to various distributed query systems and parallel computing platforms that utilize the Directed Acyclic Graph (DAG) as their computational paradigm. These systems include, but are not limited to, Spark, Flink, AsterixDB [10], Impala [13], SCOPE [17], and Myria. However, it's important for users to recognize and accommodate the subtle differences between Apache Hive and the other systems. Using Spark as an example, it is important for users to be aware that Spark's tasks are structured in such a way that there is no inter-task waiting, since consumer tasks are scheduled only after all the provider tasks have been completed. Therefore, when illustrating dependencies, researchers will need to adapt their visual design strategy, specifically by altering the axis of dependencies.

8 Conclusions

We propose QEVIS, an interactive visual analytics system for understanding distributed query execution. QEVIS incorporates a suit of views that visualize query execution at different granularities and thus allows to analyze complex query execution problems. In particular, we (i) design a new layout algorithm to compactly display the overall execution progress of the jobs in a query and their dependencies; (ii) devise two anomaly scoring methods and corresponding visualizations to show the overall anomaly degrees of the jobs and machines; (iii) propose a scatter plot-based visual encoding to summarize the massive atomic tasks and complex data dependencies among them; and (iv) implement a suite of auxiliary views and rich interactions to support cross-view exploration. We deploy QEVIS in the production environment of a company and use it to analyze queries from real applications. There are two promising future directions: (i) extending the techniques of QEVIS to other big data systems (e.g., Spark, Flink); and (ii) designing methods to automatically identify query execution bottlenecks and generate effective solutions to resolve these bottlenecks.

Acknowledgments

This work is partially supported by the National Key R&D program of China (Grant No. 2021YFB3301500), Guangdong Provincial Natural Science Foundation (Grant No. 2019A1515111047), Shenzhen Colleges and Universities Stable Support Grant (Grant No. 20200811104054002), Shenzhen Fundamental Research Program (Grant No. 20220815112848002), the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and a research gift from Huawei Guass department.

References

[1] Cloudera manager: Hadoop administration tool, 2022. https://www.cloudera.com/products/product-components/cloudera-manager.html.

[2] Apache pig, 2023. https://pig.apache.org/.

[3] dagre - graph layout for javascript, 2023. https://github.com/dagrejs/dagre.

[4] Dr. elephant - monitoring and tuning apache spark jobs on hadoop, 2023. https://github.com/linkedin/dr-elephant.

[5] Inviso, 2023. https://github.com/Netflix/inviso.

[6] Prometheus, 2023. https://prometheus.io/.

[7] Rug plot, 2023. https://en.wikipedia.org/wiki/Rug_plot.

[8] Spark web ui, 2023. https://spark.apache.org/docs/latest/web-ui.html.

[9] Tez ui, 2023. https://tez.apache.org/tez-ui.html.

[10] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, et al. Asterixdb: A scalable, open source bdms. *arXiv preprint arXiv:1407.0454*, 2014.

[11] L. Battle, D. Fisher, R. DeLine, M. Barnett, B. Chandramouli, and J. Goldstein. Making sense of temporal queries with interactive visualization. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 5433–5443, 2016.

[12] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 221–230, 2018.

[13] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. Kuff, D. K. A. Leblang, N. Robinson, D. R. S. Rus, J. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, pp. 1–10, 2015.

[14] L. Brasseur. Florence nightingale's visual rhetoric in the rose diagrams. *Technical Communication Quarterly*, 14(2):161–182, 2005.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[16] C. Cerullo and M. Porta. A system for database visual querying and query visualization: Complementing text and graphics to increase expressiveness. In *18th International Workshop on Database and Expert Systems Applications*, pp. 109–113, 2007.

[17] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[18] H. Chen, W. Chen, H. Mei, Z. Liu, K. Zhou, W. Chen, W. Gu, and K.-L. Ma. Visual abstraction and exploration of multi-class scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1683–1692, 2014.

[19] S. Di Bartolomeo, M. Riedewald, W. Gatterbauer, and C. Dunne. Stratisfimal layout: A modular optimization model for laying out layered node-link network visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):324–334, 2021.

[20] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *7th Workshop on Programmability Issues for Heterogeneous Multicores, Vienna, Austria*, 2014.

[21] A. S. Foundation. The apache hadoop project., 2023.

[22] T. Fujiwara, J. K. Li, M. Mubarak, C. Ross, C. D. Carothers, R. B. Ross, and K.-L. Ma. A visual analytics system for optimizing the performance of large-scale networks in supercomputing systems. *Visual Informatics*, 2(1):98–110, 2018.

[23] S. Gathani, P. Lim, and L. Battle. Debugging database queries: A survey of tools, techniques, and users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1–16, 2020.

[24] D. Gotz, J. Zhang, W. Wang, J. Shrestha, and D. Borland. Visual analysis of high-dimensional event sequence data via dynamic hierarchical aggregation. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):440–450, 2019.

[25] Y. Guo, S. Guo, Z. Jin, S. Kaul, D. Gotz, and N. Cao. Survey on visual analysis of event sequence data. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):5091–5112, 2021.

[26] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, pp. 881–884, 2014.

[27] H. Jaakkola and B. Thalheim. Visual sql–high-quality er-based query treatment. In *International Conference on Conceptual Modeling*, pp. 129–139, 2003.

[28] V. Jeyakumar, O. Madani, A. Parandeh, A. Kulshreshtha, W. Zeng, and N. Yadav. Explainit!–a declarative root-cause analysis engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 333–348, 2019.

[29] S. P. Kesavan, T. Fujiwara, J. K. Li, C. Ross, M. Mubarak, C. D. Carothers, R. B. Ross, and K.-L. Ma. A visual analytics framework for reviewing streaming performance data. In *2020 IEEE Pacific Visualization Symposium*, pp. 206–215, 2020.

[30] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: debugging mapreduce job performance. *arXiv preprint arXiv:1203.6400*, 2012.

[31] A. Leventidis, J. Zhang, C. Dunne, W. Gatterbauer, H. Jagadish, and M. Riedewald. Queryvis: Logic-based diagrams help users understand complicated sql queries faster. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2303–2318, 2020.

[32] J. K. Li, T. Fujiwara, S. P. Kesavan, C. Ross, M. Mubarak, C. D. Carothers, R. B. Ross, and K.-L. Ma. A visual analytics framework for analyzing parallel and distributed computing applications. In *2019 IEEE Visualization in Data Science*, pp. 1–9, 2019.

[33] H. Liu, B. Tang, J. Zhang, Y. Deng, X. Yan, X. Zheng, Q. Shen, D. Zeng, Z. Mao, C. Zhang, et al. Ghive: accelerating analytical query processing in apache hive via cpu-gpu heterogeneous computing. In *Proceedings of the 13th Symposium on Cloud Computing*, pp. 158–172, 2022.

[34] H. Liu, B. Tang, J. Zhang, Y. Deng, X. Zheng, Q. Shen, X. Yan, D. Zeng, Z. Mao, C. Zhang, et al. Ghive: A demonstration of gpu-accelerated query processing in apache hive. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 2417–2420, 2022.

[35] P. Liu, S. Zhang, Y. Sun, Y. Meng, J. Yang, and D. Pei. Fluxinfer: Automatic diagnosis of performance anomaly for online database system. In *2020 IEEE 39th International Performance Computing and Communications Conference*, pp. 1–8, 2020.

[36] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu, et al. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment*, 13(8):1176–1189, 2020.

[37] S. Malik, B. Shneiderman, F. Du, C. Plaisant, and M. Bjarnadottir. High-volume hypothesis testing: Systematic exploration of event sequence comparisons. *ACM Transactions on Interactive Intelligent Systems*, 6(1):1–23, 2016.

[38] L. Micallef, G. Palmas, A. Oulasvirta, and T. Weinkauf. Towards perceptual optimization of the visual design of scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 23(6):1588–1599, 2017.

[39] D. Moritz, D. Halperin, B. Howe, and J. Heer. Perfopticon: Visual query analysis for distributed databases. In *Computer Graphics Forum*, vol. 34, pp. 71–80, 2015.

[40] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. 1996.

[41] V. G. Pinto, L. Stanisic, A. Legrand, L. M. Schnorr, S. Thibault, and V. Danjean. Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach. In *2016 Third Workshop on Visual Performance Analysis*, pp. 17–24, 2016.

[42] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1357–1369, 2015.

[43] S. A. Sakin, A. Bigelow, R. Tohid, C. Scully-Allison, C. Scheidegger, S. R. Brandt, C. Taylor, K. A. Huck, H. Kaiser, and K. E. Isaacs. Traveler: Navigating task parallel traces for performance analysis. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):788–797, 2022.

[44] M. Sedlmair, M. Meyer, and T. Munzner. Design study methodology: Reflections from the trenches and the stacks. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2431–2440, 2012.

[45] C. Sigovan, C. W. Muelder, and K.-L. Ma. Visualizing large-scale parallel communication traces using a particle animation technique. In *Computer Graphics Forum*, vol. 32, pp. 141–150, 2013.

[46] A. Simitsis, K. Wilkinson, J. Blais, and J. Walsh. Vqa: vertica query ana-

lyzer. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 701–704, 2014.

[47] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pp. 795–806, 2010.

[48] J. Teoh, M. A. Gulzar, G. H. Xu, and M. Kim. Perfdebug: Performance debugging of computation skew in dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 465–476, 2019.

[49] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[50] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th International Conference on Data Engineering*, pp. 996–1005, 2010.

[51] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pp. 1–16, 2013.

[52] J. Wu, Z. Guo, Z. Wang, Q. Xu, and Y. Wu. Visual analytics of multivariate event sequence data in racquet sports. In *2020 IEEE Conference on Visual Analytics Science and Technology*, pp. 36–47, 2020.

[53] C. Xie, W. Xu, and K. Mueller. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):215–224, 2018.

[54] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1599–1614, 2016.

[55] X. Zhao, Y. Wu, W. Cui, X. Du, Y. Chen, Y. Wang, D. L. Lee, and H. Qu. Skylens: Visual analysis of skyline on multi-dimensional data. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):246–255, 2017.