iQCAR: Inter-Query Contention Analyzer for Cluster Computing Frameworks

Prajakta Kalmegh, Shivnath Babu, Sudeepa Roy
Duke University
{pkalmegh,shivnath,sudeepa}@cs.duke.edu

ABSTRACT

Analyzing performance interferences faced by a query due to concurrent workloads, and investigating the symptoms of these contentions is critical in order to better manage the workloads in a shared cluster. Today no tools exist to help an admin perform a deep analysis of query interaction patterns and constrains them to manually diagnose the aberrations in query schedules or task placement strategies that lead to resource contentions. This process is tedious and non-trivial, and involves debugging through a cycle of performance explanations that makes blame attributions hard.

In this paper, we present `iQCAR` - *i*nter **Q**uery **C**ontention **A**nalyze**R**, a system that models the resource interferences between concurrent queries and provides a framework to attribute blame for contentions systematically. We develop a methodology called **R**esource **A**cquire **T**ime **P**eriod (RATP) that helps us quantify the value of blame towards contentious queries accurately. Finally, we show that a cycle of query interactions can be broken to enable step-wise deep exploration of performance bottlenecks faced by a query using a multi-level directed acyclic graph called `iQC-Graph`. Our metrics for impact analysis on `iQC-Graph` enable us to generate rules for a cluster scheduler that can be used to improve the performance of recurring workloads. In our experiments, we use these rules to intervene the task placement strategies for a microbenchmark workload using TPCDS queries on Apache Spark, and show improvement in the performance of victim queries in recurring executions.

ACM Reference Format:
Prajakta Kalmegh, Shivnath Babu, Sudeepa Roy. 2018. iQCAR: Inter-Query Contention Analyzer for Cluster Computing Frameworks. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Popular data analytics frameworks like Hadoop [16], Spark [35], Teradata [3], Vertica [5], etc. enable organizations to process diverse applications in a cluster shared among multiple tenants. Users submit analytical SQL queries to these systems along with machine learning, graph analytics, and data mining queries. Long running ETL (Extract-Transform-Load) batch queries often co-exist with short interactive Business Intelligence (BI) queries. Performance variability in a cluster shared among such mixed workloads occurs as a result of inherently variable characteristics of the queries and the system [21] (*e.g.*, data skew, change in execution plans, and failure of nodes). In order to better manage such workloads, the admin controls resource allocations to reduce conflicts by partitioning resources among tenants using capped capacities [1], reserving shares [15] of the cluster, or dynamically regulating offers to queries based on the configured scheduling policies like FAIR [31] and First-In-First-Out (FIFO). Despite such meticulous measures, providing performance isolation guarantees is still challenging since resources are not governed at a fine-granularity. Moreover, the allocations are primarily based on only a subset of the resources (only CPU in Spark [35], CPU and Memory in Yarn [30]) leaving the requirements for other shared resources unaccounted for. For instance, two queries that are promised equal shares of resources get an equal opportunity to launch their tasks in Spark. However, there are no guarantees on the usages of other resources like memory, disk IO, or network bandwidth for competing queries. This usually results in performance interferences among concurrently executing tasks of queries. If a query performs poorly or misses a deadline due to an unexpected heavy contention in the system, diagnosing whether these contentions were caused by other queries of the same tenant or by queries belonging to a different tenant can help the admin identify inconsistencies in resource allocations.

Administrators have limited options today to tackle this problem. Cluster health monitoring tools [8] and application execution analysis tools [6, 9] provide an interface to inspect performance of queries. Recent works in diagnosing performance problems [22, 33] enable users to identify root causes for slow down of a query. As such, these tools can help observe symptoms of performance degradation and resource bottlenecks in a cluster. However, assessing the contribution of concurrently running queries towards causing this impact still remains challenging. A relevant performance analysis study uses a methodology called *blocked time analysis* [25] to analyze primary sources of performance bottlenecks in the cluster. Their work emphasizes the need for using resource blocked time metrics for in-depth performance analysis; however, they do not consider the role of concurrent query executions in causing these blocked times for a task, which is a focus of this paper. An insight into answering questions like 'which queries are sources and which others are victims of contentions for a specific resource on a certain host' can help an admin identify aberrations in query schedules and better manage their workloads. As an example, identifying which tenant is responsible for submitting a **noisy query**, *i.e.* a source of *'highest'* contention, can prove particularly useful to revisit the resource shares of these tenants based on their impact on other queries. Previous attempts like CPI^2 [37], that focus primarily on analyzing CPU contention between jobs using hardware counters, prove insufficient to help quantify impacts caused by multi-resource

Conference'17, July 2017, Washington, DC, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

contentions for dataflow-based query executions. Today admins have no means to analyze this impact apart from looking at individual cluster utilization logs, specific query logs, and manually identifying correlations in both. This forms the motivation of our work.

Our Contributions: In this paper, we present iQCAR - *i*nter **Q**uery **C**ontention **A**nalyze**R**, a system that formally models the resource interferences between concurrent queries and provides a framework to attribute blame for contentions systematically. iQCAR provides a robust, thorough, and extensible framework to generate multi-level systematic explanations towards the contentions faced by a query by unifying the knowledge of high-level dataflow dependencies with the low-level implementation caveats of massively parallel cluster computing frameworks. While there have been several attempts to drill-down to the systemic root causes when a query slows down, we believe that iQCAR is a first attempt towards using symptoms of low-level resource contentions to analyze high-level inter-query interactions. Specifically, we make the following contributions:

- **Explanations using iQC-Graph:** We present the user with three types of explanations that aid in deep exploration of resource interferences due to concurrent queries:
  - **Immediate Explanations:** identify disproportionate waiting times a query spends for a particular resource,
  - **Deep Explanations:** inspect this wait-time for each resource used by the query on specific cluster hosts,
  - **Blame Explanations:** investigate the contribution of concurrent queries that are responsible towards the slowdown for a particular query.

  We formally model these explanations using a multi-level Directed Acyclic Graph (DAG), called iQC-Graph, that captures the conflicts for resources at different granularity.
- **Blame Attribution:** We develop a metric called *Resource Acquire Time Penalty* (RATP) to capture the wait-time distributions for a query. Using RATP of a task as a basis, we develop a metric for assigning blame to concurrently running tasks to compute their fault in causing contention for this task. We show how this metric compares with other prevalent measures and helps us avoid false attributions.
- **Blame Analysis:** We further define a *Degree of Responsibility* (DOR) metric to assign blame or responsibility to each node in iQC-Graph for causing contention to a query under consideration. Our Application Programming Interface (API) enables administrators of cluster computing systems to perform four concrete use-cases in detecting: (i) hot resources, (ii) slow nodes, (iii) high impact *causing* noisy queries, and (iv) high impact *receiving* victim queries.
- **Rules for Cluster Scheduler:** We use the top-k explanations output at each level by our blame analysis API in generating alternative query placement and query priority readjustment rules. We also show how our rule generation module can be extended to output more heuristics to be used by a scheduler for tweaking with the workload schedules.
- **Evaluation and User-Study:** We conduct correctness, efficacy, scalability, and overhead analysis experiments to evaluate iQCAR using TPCDS queries on Apache Spark. The results of our user-study involving participation from users of different expertise helps substantiate our claims of accuracy. The study also shows how iQCAR saves users a huge time compared to the alternatives available today.
- **Web-based UI for deep exploration:** Analyzing contentions step-wise in a multi-level large scale graph is non-trivial. To this end, we provide our users a web-based front-end to navigate or explore through levels of iQC-Graph and find answers for the above use-cases.

The approach developed in iQCAR can be applied to any parallel cluster computing framework that summons for an end-to-end performance interference analysis tool for dataflow-based representation of concurrently running queries.

Roadmap. In Section 2, we define preliminary concepts, and in Section 3 we describe the key challenges in analyzing dataflows, multi-resource contentions and attributing blame in a shared cluster. We introduce RATP and present our blame attribution process in Section 4. We describe the construction of iQC-Graph in Section 5. We present our blame analysis API and show how we output rules in Section 6. Our experimental and user-study results are in Section 7. We compare our approach with related work in Section 8, and finally conclude with future work in Section 9.

2 BACKGROUND AND TERMINOLOGY

In this section, we review some concepts in cluster computing systems and introduce terminology used in the paper.

2.1 Stages, Tasks, and Dataflow Dependencies

Users of Hadoop [16] and Spark [35] submit applications through high level data processing engines, *e.g.*, SparkSQL, Dataframes API [12], GraphX [32], D-Streams [36], MLlib [24], Hive [29], and Oozie [20]. These applications are processed as a physical execution plan or a *dataflow*, which is a DAG of low-level parallelizable units (*e.g.*, map-reduce *jobs* in Hadoop, *stages* in Spark). In this paper, for simplicity, we adopt the term **stages**[1]. Each edge in the DAG represents the dataflow between these stages. A stage is composed of a **task set** where each **task** performs the same transformation in parallel on different blocks of the dataset. Each stage starts upon completion of all its parent stages in the dataflow DAG. The root stage of the DAG is the final stage, whereas the leaf stages represent the stages scanning input data. The **depth** of a stage in the query DAG is counted as the maximum number of stages on a directed path from that stage to the root stage in the DAG. We illustrate these concepts with an example:

Example 2.1. *Consider the dataflow DAG of a query Q_0 in Figure 1 comprising six stages $s_0, s_1, \cdots, s_5$. The stage s_5 is the root (output) stage of the DAG, whereas the leaves s_0, s_1, s_2 scan input data. The stage s_3 can start only when all of s_0, s_1, s_2 are completed. The depth of stage s_5 is 1, and the depth of stage s_1 is 4.*

[1]In practice, an application is decomposed into a DAG of *jobs*; An *action* of a job, defined by job boundaries, is executed using a DAG of stages. We do not include this additional layer of jobs in our model since it does not affect our approach and algorithms.

Table 1: Wait-time Metrics in `iQCAR`: `SWT` and components of `RWT` for a task

Wait Time Component	Description	Implementation
Scheduling Waiting Time (`SWT`)	Time since the stage was submitted to the scheduler queue till the time the task is launched on a host	We collect the stage submission time to the queue and a task's launch time from Spark logs)
CPU Waiting Time (`CWT`)	Time spent blocked for CPU cycles after it is launched	We instrumented Spark to capture lock wait time and blocked time from the ThreadMxBean API.
Memory Waiting Time (`MWT`)	Time the task waits until it gets the required execution and/or storage memory	We instrumented Spark to capture the time to acquire both storage memory and execution memory.
Network Waiting Time (`NWT`)	Time the task spends in reading shuffle data from the network	We collect the *fetch wait time* logs from Spark
IO Waiting time (`IOWT`)	Time that the task spends blocked on IO operation	We use *IO scan time* metric for IO reading time and *shuffle write time* metric for IO writing time

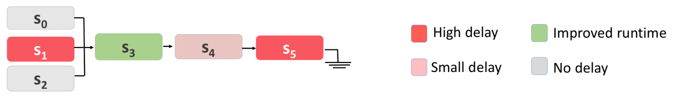
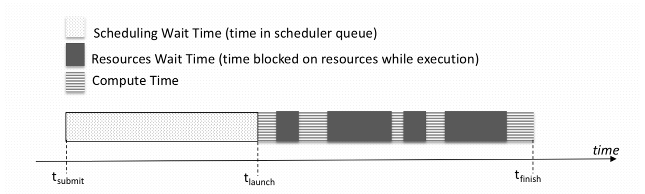


Figure 1: Dataflow DAG of an example query Q_0.

2.2 Wait-Time Metrics

Figure 2 shows the lifecycle of a task after a stage is submitted to the resource allocation module. We use `SWT` to denote the time it took to launch this task while its stage was waiting in the scheduling queue. The other components, `CWT`, `MWT`, `IOWT` and `NWT` (resp. CPU, memory, IO, network) denote the time a task is blocked on a particular resource after it is launched, and contribute to the overall *Resource Wait Time* as shown in Figure 2. Table 1 summarizes how we capture each of these wait-time components in Spark.



Scheduling Wait Time (time in scheduler queue)
Resources Wait Time (time blocked on resources while execution)
Compute Time
time
t_{submit}
t_{launch}
t_{finish}

Figure 2: Components of a stage's runtime

2.3 Target, Source, and Noisy Queries

Critical Path: A dataflow DAG may consist of many chains of stages running in parallel. We define a **critical path** as the sequence of stages with maximum overall runtime *i.e.*, whose cumulative runtime dominates the total runtime of the query. In Figure 1, stages $s_1 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$ form the critical path of query Q_0.

Target query and target stages: Any query in the system can be a ***potential*** source or a target of contention. Each of the queries chosen for deep exploration in `iQCAR` is termed as a **target query** Q_t, its stages as **target stages**, and its tasks as **target tasks**.

Source query and source stages: We refer to other concurrently running queries Q_s that can possibly cause contention to a target query Q_t as **source queries** (if the stage(s) of Q_s are waiting in the scheduler queue or running concurrently with the stage(s) of Q_t). Stages of Q_s are **source stage**, and its tasks are **source tasks**.

Noisy Queries and Aggressive Queries: A **noisy query** is a source query that causes the highest contention for the target query. Noisy queries are not necessarily *rogue*[2]; they might adhere to all resource allocation constraints imposed by the scheduler, and yet hold unaccounted resources that are also needed by another query, thus affecting its performance. A noisy query for one target query may not be necessarily noisy for another query. We refer to noisy queries that cause multi-resource contentions for several concurrent queries with high impact as **aggressive queries**.

3 CHALLENGES

We use the following example to describe the challenges that drive our choices of metrics and graph-based model.

EXAMPLE 3.1. *Suppose an admin notices slowdown of a recurring query Q_0 shown in Figure 1 in an execution compared to a previous one, and wants to analyze the contentions that caused this slowdown. The admin can use tools like SparkUI [9] to detect that tasks of stage s_1 took much longer than the median task runtime on host (i.e., machine) X, and then can use logs from tools like Ganglia [8] to see that host X had a high memory contention. Similarly she notices that s_5 was running on host Y that had a high IO contention. Further, the admin sees that stage r_3 of another query Q_1 was executing concurrently with only stage s_1 of Q_0, while stages u_5, u_7, u_9 of query Q_2 were concurrent with s_1 and s_5. Overall, stages s_1 and s_5 of Q_0 (in dark red) were a victim of contention that needs to be explored.*

Challenge 1. Analyzing Contentions on Dataflows:
Dataflow semantics should be considered when calculating and distributing blame to concurrent queries.

Today, there is no easy way for the admin to know which stage of Q_0 in Example 3.1 was responsible in the overall slowdown of this query, and whether Q_1 or Q_2 (and which of their stages) is primarily responsible for creating this contention. It is possible that even if both the target stages s_1 and s_5 spent the same time waiting for a resource (say, CPU), the amount of contention faced by them could differ if they processed different sizes of input data. Identifying such disproportionality in wait-times faced by its stages can help an admin assess the effectiveness of a schedule with more

[2]Users sometimes manipulate queries to consume more resources throughout their execution.

transparency. In Section 4, we show how iQCAR captures input data size semantics of each stage in appraising the impact values.

Challenge 2. Capturing Resource Utilizations:
Low-level resource usage trends captured using hardware counters cannot be used towards blame attribution for queries.

A typical approach to identify contention causing tasks in shared clusters is to look at the utilization patterns of concurrent tasks [37] for each resource. It is difficult to adopt this course for our purpose as capturing the low-level *resource usage* trends and associating them with the high-level abstract task entities is extremely non-trivial. This is difficult especially for resources like network (as requests are issued in multiple threads in background) and IO (requests are pipelined by the OS). This would require computing the precise acquisition and release timeline of each resource using hardware counters. Instead, in iQCAR, we use the wait-time distribution of tasks for each resource for this association. This simpler choice enables us to deduce and attribute fault as we show in Section 4.

Challenge 3. Capturing Resource Interferences:
CPU Time stolen from concurrent queries is inadequate to capture multi-resource contentions.

Detecting noisy neighbors is a well-defined problem in virtualized environments [4]. The concept of *stolen time* (time stolen by other processes from the CPU cycles of a victim process) [2] evolved with the need to quantify contentions on a cluster, but has been used only in the context of identifying processes that steal CPU time. This metric cannot be used *as is* owing to the difficulty to capture stolen time for other resources. As shown in Figure 3, tasks can use and be also blocked on multiple resources simultaneously resulting in multi-resource interferences between concurrent tasks. In example 3.1, suppose stage s_1 of Q_0 was a victim of a multi-resource contention (IO, CPU and memory) since it was the initial stage in the dataflow. The need to detect queries that steal such multiple resources from Q_0 simultaneously requires a more involved metric for blame attribution.

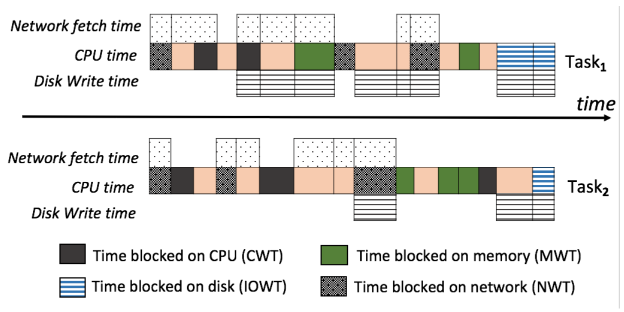

Network fetch time
CPU time
Disk Write time
Task1
time
Network fetch time
CPU time
Disk Write time
Task2
Time blocked on CPU (CWT)
Time blocked on memory (MWT)
Time blocked on disk (IOWT)
Time blocked on network (NWT)

Figure 3: Example overlap of resource usage between $Task_1$ and $Task_2$. Notice the multiplexing between the tasks for compute time.

Challenge 4. Avoiding False Attributions:
Overlap Time between tasks of a query does not necessarily signal a contention for resources between them.

Today, admins consider only the % overlap between concurrent queries to assign blame, which may lead to faulty attributions. In example 3.1, even if stage s_5 had total overlap with an IO-intensive task, the high IO contention on host Y may not have impacted s_5 since it had minimal IO activity. Such wrong attributions should be averted. Consider another scenario: concurrently running tasks on a host may belong to (i) the same stage, (ii) a different stage of the same query, or (iii) a different stage of a different query. Therefore, each task can compete either with its own *'fellow tasks'* (case (i) and (ii)) or with *'competing tasks'* (case (iii)). It is thus also important to determine whether the contentions faced by a query are on its own worth or a fault of other to avoid inaccurate accusations.

Design Goals:
There are several intricate possibilities even in this toy example, whereas there may be long chains of stages running in parallel in a real production environment making this process challenging. As a result, in order to meet the above challenges, we need a framework for contention analysis that exhibits the following properties:

- **Robust:** It should be able to capture multi-resource interferences, and distinguish between legitimate and false sources of contention. Further, it should account for other slowdown causes of a query.
- **Thorough:** It should enable a step-wise exploration of the contention space. The model should facilitate consolidation of blame and responsibility for different combinations of resources, hosts of execution, and concurrent stages and queries.
- **Extensible:** The framework needs to be extensible to plug-in additional explanation causes (*e.g.* wait-times due to garbage collection, etc) or break-down existing explanations (*e.g.* IO wait time for scanning data vs. IO wait time for writing data).

Traditional statistical approaches [11, 37] to analyze query interactions are inadequate for these goals as they fail to reason about intricate conflicts among concurrent queries and do not facilitate systematic exploration of the space. Our metrics for capturing the impact of contentions enable us to meet the goal for robustness, as we describe in the following section; whereas, the use of a graph based model for step-wise analysis provides a comprehensive and modular framework as we show in Section 5.

4 BLAME ATTRIBUTION

We develop a metric called Resource Acquire Time Penalty (RATP) that captures the distribution of wait-time per unit data processed over the execution time of task tt. We define it in Section 4.1 and discuss its application in blame attribution in Section 4.2.

4.1 Resource Acquire Time Penalty (RATP)

Definition 4.1. *In a small fixed interval $\delta\tau$, let the time spent by a target task tt waiting for resource r on host h from time τ and $\tau + \delta\tau$ be $\mathsf{WT}_{tt}^{\tau,r,h}$ and the amount of input data processed be $\mathsf{DP}_{tt}^{\tau,r,h}$. Then the **Resource Acquire Time Penalty** (**RATP**) is defined as:*

$$RATP_{tt}^{\tau,r,h} = \frac{\mathsf{WT}_{tt}^{\tau,r,h}}{\mathsf{DP}_{tt}^{\tau,r,h}} \quad (1)$$

Let τ_{tt}^{start} and τ_{tt}^{end} be the start and end times of task tt. During its runtime $T_{tt} = \tau_{tt}^{end} - \tau_{tt}^{start}$, if we assume a uniform distribution on the wait-time per unit data processed, we get,

$$\mathsf{RATP}_{tt,uniform}^{r,h} = \frac{\mathsf{WT}_{tt}^{r,h}}{\mathsf{DP}_{tt}^{r,h}} \tag{2}$$

where, $\mathsf{WT}_{tt}^{r,h}$ *is the total wait-time of task* tt *for resource* r *on host* h*, and* $\mathsf{DP}_{tt}^{r,h}$ *is the total data processed by* tt *on this host.*

As discussed in Table 4 in the appendix, the type of data processed for different resources are different. For example, consider the metric for remote bytes read over the network in Spark (REMOTE_BYTES_READ). The corresponding RATP metric, called the *'network bytes read penalty'*, is $\frac{\mathsf{NWT}}{\mathsf{REMOTE\_BYTES\_READ}}$ for a task (omitting the superscripts and subscripts), and gives us the wait time for processing one unit (one byte for our analysis) of remote data.

Features of RATP Metric: RATP exhibits the following characteristics addressing the challenges described in Section 3. (1) **Sensitive:** Since it captures the wait-time per unit of data processed, it is efficient in detecting disproportionalities in wait time discussed in Challenge 1. (2) **Computable:** RATP relies on availability of wait-times of tasks which are easier to capture at application-level [25] instead of hardware-level instrumentation (see Challenge 2). (3) **Resilient:** Depending on the frequency at which the wait-time and data usage information is collected, it can be tailored to represent complex resource interference patterns seen in Challenge 3. For instance, historical execution data can be used for recurring workloads to capture interference using RATP with more accuracy. (4) **Comparable:** The RATP value of two concurrent tasks can be compared using the area under the respective distribution curves for their overlapping period. We show next how we utilize this feature to attribute blame.

4.2 Assigning Blame using RATP

First we define slowdown of a task due to resource contention.

Definition 4.2. *Slowdown* $\mathcal{S}_{tt}$ *of a task* tt *in interval* $\delta\tau$ *between time* τ *and* $\tau + \delta\tau$ *is defined as*

$$\mathcal{S}_{tt}^{\tau,r,h} = \frac{(\mathsf{RATP}_{tt}^{\tau,r,h} - \mathsf{RATP}_{h}^{r})}{\mathsf{RATP}_{h}^{\tau,r}} \tag{3}$$

where, $\mathsf{RATP}_{h}^{\tau,r}$ *is the minimum wait-time per unit resource of* r *on host* h *in the* $\delta\tau$ *interval, e.g., for network it is* $\frac{1}{networkspeed}$.

Intuitively, it is the deviation from the ideal wait-time per-unit data on that host. The slowdown of tt will be zero when all the resources are available to the task tt. Therefore, the slowdown corresponds to the blame that can be attributed to other concurrently running tasks with tt on h, or on other quantities. Therefore, we express the slowdown defined in Equation 3 as:

$$\mathcal{S}_{tt}^{\tau,r,h} = (\sum_{1}^{n} \beta_{st\to tt}^{\tau,r,h}) + \beta_{known\to tt}^{\tau,r,h} + \beta_{unknown\to tt}^{\tau,r,h} \tag{4}$$

Here $\beta_{ss\to ts}^{\tau,r,h}$ *(discussed shortly) is the blame assigned to each of the n concurrent tasks executing on the same host during task* tt*'s execution (why their contributions can be added is derived in Appendix A),* $\beta_{known\to tt}^{\tau,r,h}$ *is the total blame value assigned to other known non-contention related causes that result in task* tt*'s wait-time, and* $\beta_{unknown\to tt}^{r,h}$ *gives the value of slowdown due to systemic issues.* Their usage can be captured via additional instrumentation or using external tools. Research in the area of diagnosing such known and unknown systemic causes for performance degradation is well established [19, 22, 33], which is out of the scope of discussion of this paper. Since one of the goals of iQCAR is to carefully attribute blame to other concurrently running tasks, below we discuss how we can compute $\beta_{st\to tt}^{\tau,r,h}$ for a source task st and target task tt.

Proposition 4.3. *The* ***blame*** $\beta_{ss\to ts}^{\tau,r,h}$ *for the contention caused by a task* st *of a source query to the task* tt *of a target query on host* h *for resource* r *in interval* τ *to* $\tau + \delta\tau$ *can be expressed as*

$$\beta_{st\to tt}^{\tau,r,h} = \frac{1}{T_{tt}} \left[\int_{o_{st,tt}^{start}}^{o_{st,tt}^{end}} \frac{\mathsf{RATP}_{tt}^{\tau,r,h}}{\mathsf{RATP}_{st}^{\tau,r,h}} d\tau \right] \tag{5}$$

where, $o_{st,tt}^{end}, o_{st,tt}^{start}$ *denote the ending and starting points of the overlap time between tasks* tt *and* st *(on host* h*), and* T_{tt} *is the total execution time of task* tt *(since a task* tt *runs on a single host, the superscripts are omitted).*

The derivation of Equation (5) is given in Appendix A.

In our implementation, to minimize the instrumentation cost (see discussion in Section 9), we assume a uniform distribution of wait-times per unit of data processed, as described earlier. Therefore, we assign blame to a source task st as :

$$\beta_{st\to tt}^{r,h} = \frac{o_{st,tt}^{end} - o_{st,tt}^{start}}{T_{tt}} \left(\frac{\mathsf{RATP}_{tt,uniform}^{r,h}}{\mathsf{RATP}_{st,uniform}^{r,h}} \right) \tag{6}$$

In systems like Spark, tasks execute on data by iterating over it in pipeline using iterator model. A uniform distribution assumption is reasonable here, especially for CPU, and Memory. Additionally, it is also a valid assumption for capturing RATP for Scheduling Slots with a FAIR scheduling policy.

Features of Blame Metric: Our quantification of blame $\beta_{st\to tt}$ enables us to meet our robustness design goal owing to the following characteristics: **(1) Avoids False Attributions:** Since the blame is computed only for periods where tt is waiting for resource r during the overlap period, the blame on st is zero if tt's wait-time is zero during the overlap. Whereas, if both st and tt are waiting for the same resource in the overlapping time (see Challenge 4), it does not qualify for blame. **(2) Accounts for known causes:** Contentions are sometimes due to processes that are part of the framework but not a valid symptom of interferences between tasks. (*e.g.*, garbage collection for CPU, HDFS replication for network). These processes impact all concurrent tasks alike for the specific resource. Equation 4 allows incorporating their impact in the $\beta_{known\to tt}$ section. **(3) Accounts for unknown causes:** Finally, slowdown could be due to a variety of other causes which are either not known or cannot be attributed to any concurrent tasks. Equation 4, accommodates all these causes into the $\beta_{unknown\to tt}$ value. Our graph-based framework, described in the next section, allows for consolidating blame for all types of causes.

5 FRAMEWORK

For any query in the workload, iQCAR presents three types of explanations that lets a user diagnose various levels of its contentions due to query interactions. These are:

Immediate Explanations (IE) *What is the impact through each resource for its slowdown?*

Deep Explanations (DE) *What is the impact through each host for a specific resource?*

Blame Explanations (BE) *What is the impact through each source query or source stage on a specific host and resource combination?*

Figure 4 shows iQC-Graph, a multi-layered directed acyclic graph, that enables us to formalize the above explanations.

5.1 Multi-Layered iQC-Graph

Level 0 and Level 1 of iQC-Graph contain the target queries and target stages respectively – these are the queries or stages that the admin wants to analyze. On the other end of iQC-Graph, Level 6 represents all source queries, while Level 5 contains all source stages, which are the queries and stages concurrently running with the target queries. The middle three levels – Levels 2, 3 and 4 – keep track of explanations of different forms and granularity that enable us to connect these two ends of iQC-Graph with appropriate attribution of responsibility to all intermediate nodes and edges. Levels 2, 3, and 4 in iQC-Graph respectively consist of *Immediate Explanations* (IE), *Deep Explanations* (DE), and *Blame Explanations* (BE); together these levels form the **Explanations Layers** of iQC-Graph. We summarize all levels of iQC-Graph in Table 2 for a quick reference. Note that it is important to separate the queries and their stages in two different levels at both ends of iQC-Graph: if multiple stages of a source query Q_s run in parallel with multiple stages of a target query Q_t, Q_s may have a higher impact on the performance of Q_t compared to other source queries.

Vertex Contributions: For each node u in the graph, we assign weights, called **V**ertex **C**ontributions denoted by VC_u, that are used later (described in Section 6) for analyzing impact. Specifically, VC_u measures the standalone impact of u toward the contention faced by t-stage. The VC values of different nodes are carefully computed at different levels by taking into account the semantics of respective nodes. For Level 0 target query, Level 5 source stage, and Level 6 source query vertices, we currently set $\mathsf{VC}_u = 1$ for all nodes u in each level. For the Level 1 target stage vertices, we set this to the cumulative CPU time (*i.e.*, the work done) by stage s in order to assign higher impacts through stages that get more work done. We now show how we compute these values for explanation nodes.

5.2 Explanation Levels:

Each node u in Levels 2, 3 and 4 is called an **explanation node**, and provides an **explanation** ϕ_v for the contention faced by a target stage. An explanation takes the following generic form:

Definition 5.1. *An* **explanation** ϕ *in* iQCAR *is a 5-tuple:* $\phi = \langle$id, type, desc, t−stage, DOR$\rangle$, *where* id *denotes the unique identifier of* ϕ *(e.g., a unique node identifier of* iQC-Graph*);* type $\in$ {IE, DE, BE} *denotes the type of the explanation* ϕ*;* desc *denotes the textual description of* ϕ *;* t−stage *denotes the target stage being explained by* ϕ*;* DOR *denotes the* Degree Of Responsibility *of* ϕ *(defined in Section 6.1), which is the cumulative measure of responsibility of* ϕ *toward the interference faced by the target query due to concurrency.*

Table 2: Levels in iQC-Graph: Levels ℓ_2,ℓ_3,ℓ_4 represent the explanation types supported in iQCAR

Level	Vertex Type	Description
ℓ_0	Target Query	Set of queries to be analyzed
ℓ_1	Target Stage	For each query at ℓ_0, stages to be analyzed
ℓ_2	Immediate Explanation	For each stage at ℓ_1, cumulative RATP for each resource
ℓ_3	Deep Explanation	For each IE at ℓ_2, cumulative RATP for each host
ℓ_4	Blame Explanation	For each DE at ℓ_3, blame towards each stage concurrent on the same host.
ℓ_5	Source Stage	The *source stage* running concurrently with *target stage*
ℓ_6	Source Query	The corresponding *source query*

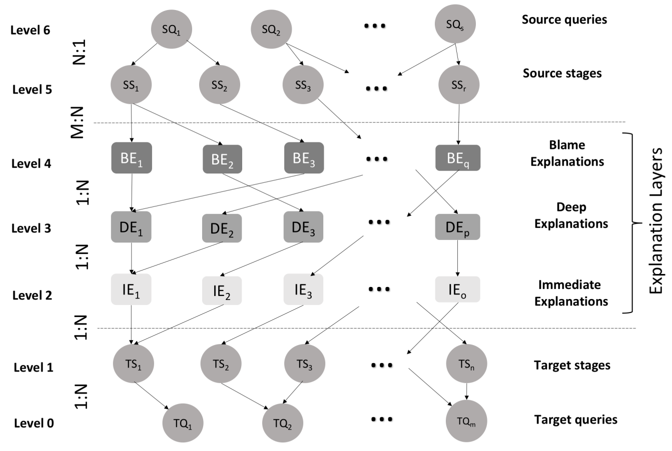


Figure 4: iQC-Graph: A multi-level Explanations Graph. The one-to-many, many-to-one, and many-to-many relationships between two consecutive layers are shown as $1 : N$, $N : 1$, $M : N$ respectively.

Since every explanation ϕ explains a particular target stage t-stage, the subgraph formed by the explanation nodes for one target stage at Level 1 is disjoint from the subgraph formed by the nodes for another target stage. They are connected back at Level 5 if multiple target stages execute concurrently with the same source stage. This property enables us to construct the subgraphs from Level 0 to Level 4 in parallel for each target query to be analyzed, thus reducing the graph construction and analysis time significantly as we see in Section 7. Next we describe how we add explanation nodes and assign node weights to them - a property used later to update the DOR values (see Section 6).

5.3 Immediate Explanations (IE)

For each target stage in Level 1, we add five nodes in Level 2 corresponding to each of the resources, namely scheduling slots, CPU, network, IO and memory. For each IE node u, we compute its VC_u as follows:

$$\mathsf{VC}_u^{\ell_2} = \sum_{tt \in \mathsf{t-stage}} \int_{\tau_{tt}^{start}}^{\tau_{tt}^{end}} \mathsf{RATP}_{tt}^{\tau,r,h} d\tau \qquad (7)$$

where τ_{tt}^{end}, τ_{tt}^{start} denote the start and the end time of a task tt of the stage `t-stage`. $\mathsf{RATP}_{tt}^{\tau,r,h}$ is computed as per Equation (1) given in Section 4. Note that the node u refers to a resource r and a task tt runs on a single host h, so the superscripts r, h do not appear on the lefthand side on the above equation. The value of $\mathsf{VC}_u^{\ell_2}$, thus, gives us the cumulative wait-time per unit of data processed by stage `t-stage` for the specific resource represented by the `IE` node.

Example 5.2. `IE`*s* ***in Example 3.1:*** *Suppose the user selects Q_0 as the target query, and wants to analyze the contention of the stages on the critical path. First, we add a node for Q_0 in Level 0, and nodes for s_1, s_3, s_4, s_5 in Level 1. Then in Level 2, for each of these four stages on the critical path, the admin can see five nodes corresponding to different resources. Although both s_1 and s_5 faced high contentions, using* `iQC-Graph` *the admin can understand questions such as whether the memory contention faced by stage s_1 was higher than the IO contention faced by stage s_5.*

5.4 Deep Explanations (DE)

Deep explanations unfold the components of wait time distribution of a target stage `t-stage` further to keep track of the wait-time distributions per unit of data processed by a stage for a specific resource on each host of execution. First, we find all the hosts that were used to execute the tasks from `t-stage`. Then, for each `IE` node in Level 2, we add multiple `DE` nodes in Level 3 corresponding to these hosts. The value of $\mathsf{VC}_u^{\ell_3}$ is set similar to Equation 7 except that the summation is done over all tasks of `t-stage` executing only on host h in question. Specifically,

$$\mathsf{VC}_u^{\ell_3} = \sum_{tt \in h-tasks} \int_{\tau_{tt}^{start}}^{\tau_{tt}^{end}} \mathsf{RATP}_{tt}^{\tau,r,h} d\tau \tag{8}$$

where, τ_{tt}^{end}, τ_{tt}^{start} denote the start and the end time of a task tt of the stage `t-stage`, node u corresponds to resource r and host h, and $h - tasks$ denotes the list of tasks in `t-stage` executing on host h.

Example 5.3. `DE` ***in Example 3.1:*** *Suppose only the trailing tasks of stage s_5 executing on host Y faced IO contention due to data skew. Using* `iQC-Graph`*, a user can analyze the* `RATP` *on host Y and find that the wait-time per unit data was much less compared to the average* `RATP` *for tasks of stage s_5 executing on other hosts. This informs the user that the slowdown on host Y for tasks of s_5 was a straggler problem owing to skew.*

For every `IE` node in Level 2 corresponding to resource r, we add $P_r \times H$ new nodes in Level 3, where P_r is the number of different requests that can lead to a wait time for resource r, and H is the number of hosts involved in the execution of `t-stage`. For instance, the IO bottleneck of `t-stage` can be explained by the distribution of time spent waiting for `IO_READ` and `IO_WRITE`. Therefore, for r = IO, $P_r = 2$, and for each host we add two nodes for `IO_BYTES_READ_TIMEPERIOD` and `IO_BYTES_WRITE_TIMEPERIOD` in Level 3.

Cumulative VS Max Values: Note that we consider the cumulative `RATP` values for vertex contributions in Equation 7 and Equation 8. The other alternative approaches include considering *max* or *average* values for the tasks in a stage; however, *sum* captures the **total cluster time** (similar to the notion of *Database Time* in [18]) spent on waiting for resources per unit data by each stage and, thus, enables us to analyze the overall slowdown of a stage in the cluster. This enables us to compare disproportionalities in the wait-times of two stages (or queries) irrespective of their degree of parallelism in the cluster.

5.5 Blame Explanations (BE)

To create Levels 4, 5, and 6 of `iQC-Graph`, first we find all source stages that were concurrent with `t-stage` in Level 1; we add these source stages in Level 5 and the source queries they belong to in Level 6. Then, we connect the nodes in Level 5 with the nodes in Level 3 (`DE`) by creating new Blame Explanations (`BE`) nodes in Level 4 as follows. For each `DE` node v in Level 3 corresponding to `t-stage`, host h, and type of resource request r, if `t-stage` was executing concurrently with P source stages on host h, we add P nodes u in Level 4 and connect them to v. The `VC` of each `BE` node is then computed from the blame attribution of all target stages in Level 3 that it can potentially affect:

$$\mathsf{VC}_u^{\ell_4} = \sum_{tt \in h-tasks} \beta_{st \rightarrow tt}^{r,h} \tag{9}$$

where $h - tasks$ is the list of tasks executing on host h, and node u corresponds to a source task st, resource r, and host h. The value of $\beta_{st \rightarrow tt}^{r,h}$ is computed as described earlier in Equation (6) given in Section 4.

Example 5.4. `BE` ***in Example 3.1:*** *Since we know that stage r_3 of source query Q_1 was executing concurrently on machine X with stage s_1 of Q_0, for each* `DE` *vertex corresponding to machine X, we add one* `BE` *vertex to capture the concurrency from r_3. Next, since stages u_5, u_7, u_9 of another source query Q_2 were running on machine Y, for each* `DE` *vertex corresponding to machine Y we add three* `BE` *vertices in Level 4 for this concurrency.*

Capturing the Unknowns: As mentioned in Section 4, a query may experience slowdown due to myriad reasons including concurrent executions, system issues (hardware and software), the query's own code and design issues etc. To capture such known and unknown causes in our model, we add a node for 'Unknown' `BE`s in `iQC-Graph`. Currently, we attribute all blame that cannot be assigned to any concurrent query execution to this node, however, it is easy to extend the framework to add any measurable and known causes (like wait-time due to garbage collection) and attribute that blame to a "Known" `BE` vertex.

Next, we discuss how the graph-based framework enables us to consolidate the explanation values by defining two metrics for contention analysis, namely Impact Factor (`IF`) and Degree of Responsibility (`DOR`).

6 EXPLORATION WITH IQC-GRAPH

In this section, we describe how the choice of a graph-based model enables us to consolidate the contention and blame values for a systematic deep exploration.

6.1 Metrics for Impact Analysis

While the *blame* value computed using Equation 5 gives the impact a source task st has towards the wait-time of a target task tt, we need other measures to compare the contentions at various levels of `iQC-Graph`. To achieve this, we compute two additional measures:

Impact Factor (IF) of edges and cumulative *Degree of Responsibility* DOR for every node in iQC-Graph.

Impact Factor (IF): Once the Vertex Contributions VC_u of every node u in iQC-Graph is computed to estimate the standalone impact of u on a target stage, we compute the Impact Factor IF_{uv} on the edges (u, v). This enables us to distribute the overall impact received by each child node v among its parent nodes u-s. For instance, IF_{uv} from a DE node u to an IE node v gives what fraction of total impact on an IE node can be attributed to each of its parent DE nodes. Figure 5 shows an example of the impact received by node u_1 from nodes v_1, v_2, v_3. The details on its computation and the algorithm can be found in Appendix B.1.

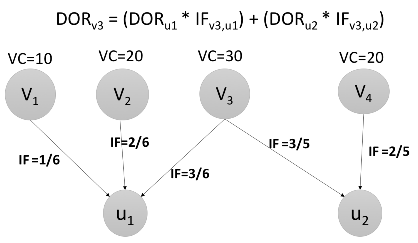

$\text{DOR}_{v3} = (\text{DOR}_{u1} * \text{IF}_{v3,u1}) + (\text{DOR}_{u2} * \text{IF}_{v3,u2})$
VC=10
VC=20
VC=30
VC=20
V_1
V_2
V_3
V_4
IF=1/6
IF=2/6
IF=3/6
IF=3/5
IF=2/5
u_1
u_2

Figure 5: Example shows how VC values are used to compute IF of each edge, and how IF s are used to compute DOR values of each node.

Degree of Responsibility (DOR): Finally we compute the Degree of Responsibility DOR_u of each node u mentioned in Definition 5.1, which stores the *cumulative impact* of u on a target query t. The value of DOR_u is computed as the *sum of the weights of all paths* from any node u to the target query node t, where the weight of a path is the product of all IF_{vw} values of all the edges (v, w) on this path. If we choose more than one query at Level 0 for analysis, a mapping of the values of DOR toward each query is stored on nodes at Level 5 and 6. The computation of DOR of node v_3 is illustrated in Figure 5. It can be noted that our framework also allows for consolidating impact originating from 'Unknown' explanation nodes at each level, enabling an admin to rule out slowdown due to resource interference issues if the impact through these nodes is high. We give the details of its computation, the algorithm, and runtime analysis in Appendix B.2.

6.2 Algorithms for Contention Analysis

In this section, we discuss three applications of iQCAR that can aid deep exploration of contention symptoms in a shared cluster.

6.2.1 Finding Top-K Contentions for Target Queries. iQCAR outputs the relevant top-k explanations (IE, DE, BE), or the top-k source stages and queries with a high cumulative impact (DOR) for a given value of k. The admin can choose to halt at particular level of analysis or may unfold all levels of explanations up to the source queries to get a more detailed narrative. An interesting observation is that, even if we are interested only in the top-k nodes at the highest level of iQC-Graph, still we cannot prune the nodes with low DOR values at lower levels, since there may be many paths through lower valued nodes from a lower level to a higher level (see Figure 6). **Running time.** Given the iQC-Graph, the top-k vertices at *all* levels can be found using the linear time selection algorithm [14] since the nodes at different levels are disjoint.

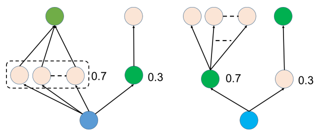


Figure 6: Nodes with high weights at a higher level having paths from nodes with low weights at a lower level

6.2.2 Detecting Aggressive Source Queries. iQCAR allows the admin to do a top-down analysis on a source query or source stage to explore how it has caused high impact to all concurrent queries. To detect such aggressive queries, we find the (top-k) Level 6 nodes having the highest value of total DOR toward all affected queries (recall that for each source query, we keep track of the DOR value toward each target query).

6.2.3 Identifying Slow Nodes and Hot Resources. There are three ways an admin can analyze contentions through hosts or resources. *First*, performing a top-k analysis on IE or DE levels will yield the hot resource (IE) and its corresponding slow node (DE) with respect to a particular target query. *Second*, to find the instances of top contentions between any source and any target query, the admin can query the top-k paths with maximum weight (product of IF on their edges). Appendix B.3 gives the algorithm. *Third*, in order to get the overall impact of each resource or each host on all target queries, iQCAR provides an API to (i) *detect slow nodes*, *i.e.*, group all nodes in Level 3 (DE) by hosts, and then output the total outgoing impact (sum of all IF values) per host, and (ii) *detect hot resources*, *i.e.*, - output the total outgoing impact per wait-time component nodes in Level 2 (IE).

6.3 Rules for Cluster Scheduler

We present two types of rules that can be used by a cluster scheduler for recurring executions of workloads: (a) alternate query placement and (b) query priority readjustment. Our rule generator uses the blame analysis API described in previous section to output Json rules. To test their efficacy, we extended the Spark scheduler to parse and apply a selected rule from this file.

Definition 6.1. *A* **rule** $\mathfrak{R}$ *in* iQCAR *is a 6-tuple:* $\mathfrak{R} = \langle \text{id}, \textit{rule-type}, \textit{query-id}, \textit{query-type}, \textit{rule-value}, \textit{rule-order} \rangle$*, where* id *is the unique identifier of* $\mathfrak{R}$*, rule-type is one of the types discussed below, query-id is the query for which the rule is applicable, query-type denotes whether it is a source or target query, rule-value is used by the scheduler to quantify the rule based on the rule-type, rule-order is the order in which to apply rules of the same rule-type.*

(1) Alternate Query Placement ($\mathfrak{R}$ – *AGG*): In this type of rules, we output the top-k aggressive queries identified by iQCAR using the approach presented in Section 6.2.2. The query-id is the id for this query, and query-type is set as 'Source'. Finally, the rule-value is set to be the maximum share this query-id was entitled to in a FAIR allocation policy, *i.e.* $\frac{1}{n}$ where n is the number of maximum query concurrency in the workload. Our extension to Spark scheduler takes the rule `id` as input and if it matches the type $\Re - AGG$, it places the corresponding query in a dedicated pool with its share set to rule-value.

(2) Query Priority Readjustment ($\Re - DYNP$): The second type of rules generates two types of priority rectification suggestions: (i) for the top-k affected target queries that suffered the highest impact, we generate k rules for each query-id. The rule-value is then set to $rule{-}value = P_{orig} + i$ where P_{orig} is the priority of the corresponding query-id in the previous workload, and i is the rank of the queries output in the top-k analysis. Owing to a priority higher than its previous execution, each of the query's tasks get more opportunity to be launched. Similarly, (ii) for top-k impacting source queries, the rule outputs new priorities that are lower than the priority each query had in its previous execution.

Currently, `iQCAR` applies the above rules one at a time. We show through our experiments in Section 7 how application of these rules creates interventions in the original schedule that benefit the affected queries.

7 EVALUATION

Our empirical evaluation of `iQCAR` consists of the following types of experiments: :

- **Correctness Experiments** (Section 7.2): In these experiments, we verify that `iQCAR` outputs the correct ordering of explanations at each level. We also show that `iQCAR` reacts to induced contention conditions and diagnoses the anticipated symptoms subsequently
- **Efficacy Experiments** (Section 7.3): The goal of these experiments is to show how the rules output by `iQCAR` can be used by a cluster scheduler to intervene with the workload schedule in recurring executions to improve query runtimes.
- **Comparison Experiments** (Section 7.4): The purpose of these experiments is to demonstrate that the mere use of overlap-time between concurrent queries is inadequate towards blame attribution.
- **Scalability Experiments** (Section 7.5): We use synthetically generated data to construct `iQC-Graph` for a varying number of nodes to showcase the scalability of our graph construction and analysis algorithms.

Finally, we present our results of a user-study (Section 7.6) that was geared towards demonstrating how `iQCAR` can be used by an admin to analyze query contention symptoms and detect noisy neighbors contributing towards these interferences.

7.1 Experimental Setup

The rest of our experiments were conducted on Apache Spark 2.1 [35] deployed over a 10-node local cluster. Spark was setup to run using the standalone scheduler in `FAIR` scheduling mode [34] with default configurations. Each machine in the cluster has 8 cores, 16GB RAM, and 1 TB storage. A 100 GB TPC-DS [10] dataset was stored in HDFS and accessed through Apache Hive in Parquet [7] format. The SQLs for the TPC-DS queries were taken from the implementations in [28] without any modifications.

Baserun (`BASE`) for Experiments: Here we present our simulation of a real workload from `Company ABC` (that wishes to remain anonymous) by (i) selecting TPC-DS queries that reflect the resource requirements, cluster utilization and query throughput values of their workload, (ii) matching the submission time of our benchmark queries with their query arrival pattern, and (iii) scaling down the total number of queries and maximum concurrency at any time to fit the limitations of our cluster size. As such, we generate a `BASE` run and load the logs to construct `iQC-Graph`. Figure 7 shows the % hit each query took compared to its unconstrained execution.

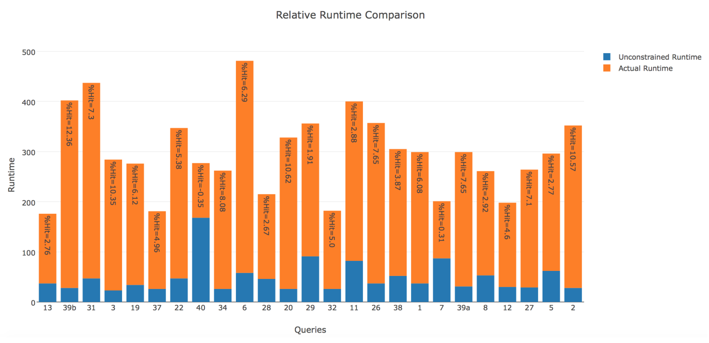


Figure 7: Q_{39b} which took the maximum hit (12.36%) is our target query Q_t. For induced contention experiments, we choose Q_7 which took the least hit (0.31%) as Q_t.

Contentions in `BASE`: `iQCAR` recommends a target query for deep exploration that takes the most hit compared to its unconstrained execution time (for example, TPC-DS query Q_{39b} takes a 629% hit in `BASE`). `iQCAR` outputs top$k$ noisy queries (for $k = 3$, we get Q_5, Q_{32}, Q_{12} in-order of decreasing `DOR` in `BASE` run) and one aggressive query (Q_4 in our sample run). For simplicity, we refer to our target query from the `BASE` run Q_{39b} as Q_t, and the top-3 noisy queries (Q_5, Q_{32}, Q_{12}) as Top_1, Top_2 and Top_3.

7.2 Correctness Experiments

7.2.1 Source Elimination. To verify that Top_1, Top_2 and Top_3 indeed cause contention to Q_t in a decreasing order, we execute three runs, namely Run-Top_1, Run-Top_2, and Run-Top_3, by eliminating each of them in order from the `BASE` schedule. Figure 8 shows the improvement in the runtime of Q_t and also the overall gain for all queries. Removing each query naturally results in reduced contention for Q_t, however, the consequential gain in runtime gradually decreases as we pick a lesser noisy query for our elimination. While this is not a practical solution to handle noisy neighbors, the purpose of this experiment is to show that they cause contention in a decreasing order, as detected by `iQCAR`.

7.2.2 Induced Contention. We tested the effects of a targeted contention on a different query from our initial `BASE` run that originally remained unaffected by concurrent query interferences. Query Q_7 in `BASE` suffered the least hit compared to its unconstrained execution (only 20% hit). Due to space constraints, we present the results for only the CPU contention effect next.

We introduce a CPU-intensive query Q_c expecting it to cause CPU contention to our target query Q_t. In order to isolate a host for inducing CPU contention, we suppress the impact due to IO and network interferences. Our induced query Q_c calculates multiple

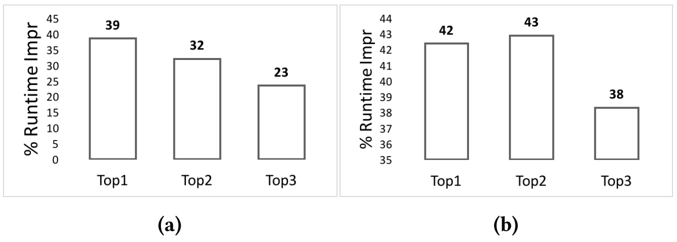


Figure 8: Source Elimination: An example showing query Top_2 is a noisy query causing contention to multiple concurrent queries ((b) shows eliminating Top_2 gives the highest benefit overall); whereas eliminating Top_1 (a) gives the highest benefit to Q_t.

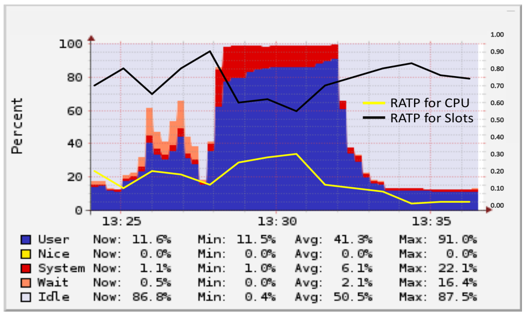


Figure 9: Induced Contention: Shows the dip in the aggregate value of SLOTS_OFFERED_TIMEPERIOD for all tasks launched on host X during our induced CPU contention window. We can see a rise in the aggregate RATP for CPU for all tasks of Q_t too.

SHA hash values over parallized collection of large strings in a loop with configurable number of iterations. The strings are embedded into the code to avoid any disk contention. All SHA hash values are discarded and only a success or failure response is returned. This is to ensure that Q_c will have a single stage in its DAG thus avoiding possibility of any network contention. We launch Q_c with a dedicated executor (`spark.executor.instances` parameter in Spark) such that all of its tasks run on the same single host of Q_t's execution. Figure 9 shows a higher CPU utilization on this host during our experiment using the Ganglia [8] time-series plot. A rise in CWT value on this host with a corresponding decline in SWT value during the contention timeline verifies that iQCAR indeed identifies this new resource bottleneck accurately.

7.3 Efficacy Experiments

As discussed in Section 5, iQCAR currently supports two types of rules to be used by the cluster scheduler (a) alternative query placement strategies, and (ii) query priority adjustments. In this section, we present experiments that demonstrate the efficacy of each type of these rules in improving the runtime of affected queries in recurring executions.

7.3.1 Alternate Query Placement. For this experiment, we use the $\Re - AGG$ rules described in Section 6.3 and apply the rule for the top-most aggressive query. In our BASE run, iQCAR identifies Q_6 as the query-id for the $\Re - AGG$ rule, hereafter termed as Q_a. We denote the set of all queries that were a victim of Q_a in the BASE run as TQ_{list}. Figure 10 compares the runtime of all queries in TQ_{list} with their new runtimes observed after our intervention. The application of our $\Re - AGG$ type rule shows significant improvement (69%) for the top affected query (Q_{22}) in BASE. We also succeed in improving the runtimes of most other affected queries, especially the ones with high impact in BASE. Q_{38} and Q_{29} did not gain much from this strategy as they both were in their last stages of execution with a few trailing tasks when Q_a was launched.

Figure 10 shows that the change in DOR of Q_a towards each affected query post intervention. Q_1 and Q_{19} who previously were the victims of Q_a, did not overlap with the contentious stages of Q_a after intervention. As a result of these reduced DORs, query Q_a is no longer the aggressive query. For some queries, the DOR of Q_a increased towards them (particularly Q_{26}) denoting that its contribution towards causing them contention was higher compared to BASE, but it still resulted in gain for Q_{26}.

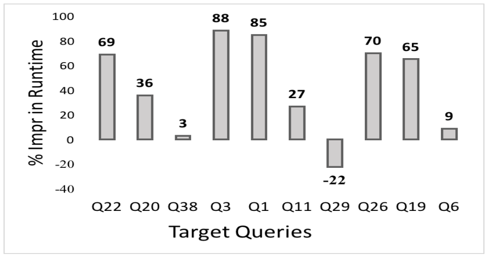
7.3.2 Priority Readjustments. Next, we test the efficacy of applying the priority amend rules $\Re - DYNP$ output by iQCAR as discussed in Section 6.3. Here, we choose to share the experiment for adjusting the priority of a single target query due to space constraints. We note that SWT was the primary contention symptom observed for our original target query Q_t (in BASE it was Q_{39b}). Accordingly, we intervene with the workload schedule by increasing the priority of Q_t in the recurring execution. We expect to see that SWT of Q_t indeed decreases as a result of our intervention.

Figure 11b shows about 56% reduction in the SWT for Q_t after our intervention, along with minor reductions in other wait-time components. This, however, leads to increased CPU contention. Despite this, we saw an overall 53% improvement in the runtime of Q_t and a 31% improvement in total runtime of all queries as shown in Figure 11a. In Figure 12, the decrease in the fraction share of SWT in the cumulative runtime of Q_t, corroborates this analysis.

7.4 Comparison Experiments

We evaluate blame attribution in iQCAR by comparing it with an alternate approach, *simple-overlap*: blame value of a source query is proportional to its overlap time with target query in the period of contention. Highest blame is attributed to query with most overlap. For these experiments, we use a simpler workload consisting of three queries $Q_1, Q_2, and Q_3$ to be able to demonstrate the limitations of *simple-overlap* approach in analyzing contentions for analytical workloads. In each repetition of an experiment, we increase the delay between the start times of queries Q_1, and Q_3; whereas Q_2 is always triggered at the same time as Q_1. Queries Q_1 and Q_2 read a large fact table (40 GB) and perform a simple group by aggregation. Q_3 reads a different fact table (31 GB) twice and performs self-join followed by a simple group by aggregation.

Figure 13 compares the DOR values with *simple-overlap*. As the delay in Q_3's start time increases, its overlap with IO stage of Q_1 reduces. In *simple-overlap* approach, this increases Q_2's contribution of IO contention. While this seems intuitive, it does not account for an important aspect - Q_2 is reading the same table as Q_1 and mostly reading the same blocks on that host. Hence its increase in contribution towards contention should not be high. This behavior is reflected in the DOR values of iQCAR which is sensitive to actual resource usage. Therefore, though the overlap time of Q_3 decreases it still creates greater IO contention for Q_1 than Q_2.



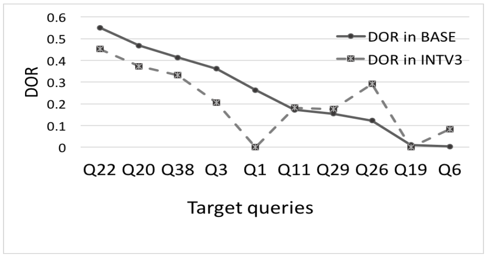


Figure 10: $\Re - AGG$: Impact of placing Q_a in a different queue. (a) shows improvement in runtime of most queries that were victims of contention caused by Q_a in BASE. The queries are ordered by the highest impact received in BASE. (b) shows a decrease in the DOR values for Q_a towards each of these queries after our intervention of applying alternate placement rule.

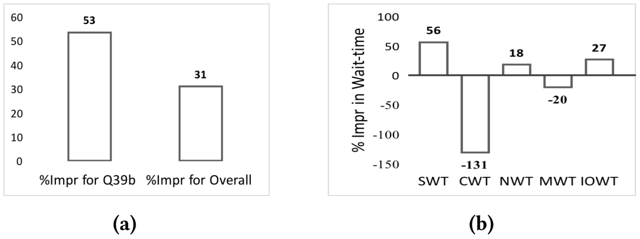


Figure 11: $\Re - DYNP$: (a) Impact of increasing priority on Q_t's performance and total runtime of all queries. (b) SWT for Q_t reduces significantly as tasks get opportunity to launch faster. iQCAR also captures increased CPU contention as a result of this increase in parallelism.

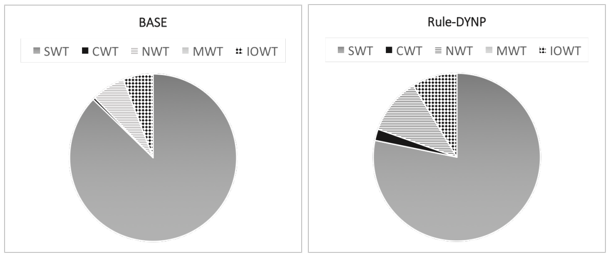


Figure 12: $\Re - DYNP$: Share of each wait-time component in total cumulative wait-time of Q_t. Fraction of total contention due to SWT reduces as a result of increased priority of Q_t.

7.5 Scalability Experiments

Although iQCAR is an offline system, the analysis should be fast enough for the admins to get a timely insightful value. Due to the limitation of our cluster size, to test the scalability of iQC-Graph we created a synthetic dataset with the same structure and dependencies with random values for node and edge weights at each level. Figure 14 shows the time taken (in log scale) to (i) construct the graph, traverse the graph for updating DOR values, and (ii) execute our top-k analysis algorithms.

7.6 User Study

We performed a user study involving participation from users with varying levels of expertise in databases, Spark and in using system monitoring tools like Ganglia. Briefly, we categorize our users based

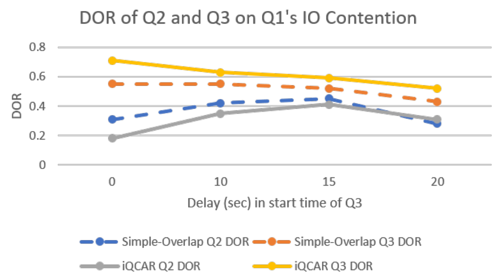


Figure 13: iQCAR vs *Simple-overlap*: Compares the blame of Q_2 and Q_3 towards the IO contention faced by Q_1.

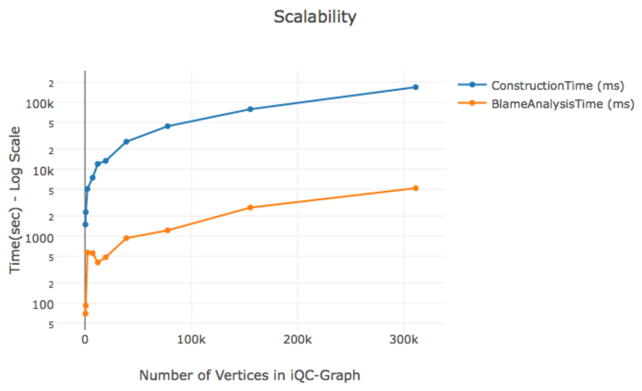


Figure 14: Scalability of iQC-Graph for an increasing number of vertices. Each dot represents the number of target queries (range 1-10) in Level-0 corresponding to the ten points on the lines.

on their combined experience in all three of these categories as (a) *Novice* (e.g. undergrad or masters database student or users with $\leq$ 1 yr experience), (b) *Skilled* (e.g. industry professionals with 1-3 yrs experience) and (c) *Expert* (e.g. DB researcher or industry professionals with more than 3 yrs experience). As such, we had 4 *Experts*, 2 *Skilled* and 4 *Novice* users. This web-based tool enabled the users to analyze a pre-executed micro-benchmark to answer an online questionnaire consisting of 10 multiple-choice questions with mixed difficulty levels. We categorized our questions as *Easy*, *Medium* and *Hard* based on the number of steps involved and the time it took to answer them manually by one of our *Experts*. At the beginning of study, each user was randomly assigned 5 questions to be answered with manual analysis using existing monitoring tools [8, 9]. The users were then shown the iQCAR user-interface (UI) to answer the rest of the 5 questions. Each question was presented progressively and the time taken to answer it was recorded in order to distinguish between genuine and casual participants. Note that in the iQCAR UI, we purposefully refrained from showing the users the final answers generated by iQCAR; instead, we presented them with plots based on (a) the measured wait-time and data processed values and (b) consolidated DOR values at each level that aided the users in selecting their answers using iQCAR tool. This activity enabled us to match the answers generated from iQCAR with those selected by our *Expert* users with manual analysis.

User-Study	Novice		Skilled		Expert	
	Manual	iQCAR	Manual	iQCAR	Manual	iQCAR
Easy	28	80	0	100	0	75
Medium	14	40	0	66	16	60
Hard	16	70	0	66	33	66

Table 3: Percent correct answers - Manual vs iQCAR

In total, users attempted 54% of our questions requiring manual analysis since they gave up on the remaining questions, whereas they answered 98% of the questions using iQCAR interface. They got 37% answers correct with manual analysis, and 69% right using iQCAR. Table 3 summarizes the results based on levels of expertise. It shows that using only visual aids on impact values and resource-to-host RATP heatmaps, the accuracy of selecting correct answers increased for each type of user compared to manual analysis. One interesting observation from our user-study is that once the users were finally given an opportunity to compare and review their chosen answers with those generated by iQCAR supplemented with a reasoning, they agreed to accept the output generated by iQCAR for 90% of the mis-matched answers. In our results, the time taken by the users in answering the questions manually was about 19% lesser than the time they took to answer using iQCAR UI. When we explored further to understand this counter-intuitive result, we noticed that irrespective of user expertise, most of them spent significant time in answering the first one or two questions but then gave up on the answers too easily for later questions. Whereas, they spent time in exploring the iQCAR UI to attempt more answers.

8 RELATED WORK

In this section we compare iQCAR with various related research projects: ***(1) Blocked Time Analysis:*** A recent study used *blocked time* metric [25] for analyzing performance of workloads on cluster computing frameworks. Their study considers the time a task spends blocked on CPU, Network and IO. The IE level of iQCAR is inspired from their approach, but we extend it to study memory contentions and time spent waiting for slots in the scheduler queue as well. More importantly, iQCAR defines a new metric (RATP) that uses the *blocked time* per resource to analyze contentions due to concurrent queries. ***(2) Analyzing query interactions:*** In [11], they show how query interactions can impact database system performance significantly. Unlike [11], we do not require any input on query type models to do performance analysis. Moreover, the focus of our paper is analyzing concurrent executions for symptoms of contentions. ***(3) Blame Attribution and Causal Monitoring:*** Causality based monitoring tools like DBSherlock [33] use causal models to perform a root cause analysis. DBSherlock analyzes performance problems in OLTP workloads. Since the motive of iQCAR is to focus on contention problems due to concurrent queries, the methodology used in DBSherlock may not be suitable to adopt. Another recent work CPI^2 [37] uses hardware counters for low-level profiling to capture resource usage by antagonist queries while the CPI (CPU Cycles-Per-Instruction) of the victim query takes a hit. Since this approach does not capture multi-resource contentions at application-level, it suffers from finding poor correlations when queries are not compute-intensive. Blame attribution has also been studied in the context of *program actions* [17]. ***(4) Performance diagnosis tools:*** Performance diagnosis has been studied in the database community [18, 33], for cluster computing frameworks in PerfXPlain [22], and in cloud based services [26]. PerfXplain uses a decision-tree approach to provide a debugging toolkit to analyze the performance of MapReduce jobs. However, it fails to consider dataflow dependencies and workload interactions. Moreover, low-level job diagnosis predicates may not be useful for a long multi-stage application unless they diagnose each job individually and find correlations in the collected data. ADDM [18] defines a notion of *Database Time* of a SQL query and they use this metric for performing an impact analysis of any resource or activity in the database. At each level of analysis, they only consider the components of this database runtime and drill-down to lower levels that consumed a significant portion of this database time. iQCAR furthers this approach to provide an end-to-end query contention analysis platform. While these studies help identify some causes for query slowdown, they do not enable an admin to analyze the deep reasons of this slowdown, which is a focus of iQCAR. ***(5) Other work on explanations in databases:*** In the context of analyzing traditional database query answers, explanation has been studied in many different contexts like data provenance [13], causality and responsibility [23], explaining outliers and unexpected query answers [27, 31], etc. The problem studied and the methods applied in this paper are unrelated to these approaches.

9 CONCLUSION

In this paper, we proposed a model that uses distributions of wait-time per unit data processed by tasks as a basis for formalizing blame attribution to concurrent queries. We further showed how our graph-based framework allows for consolidation of blame across multiple levels of explanations allowing an admin to explore the contentions and contributors of these contentions systematically. Finally, we illustrated how the top-k explanations and rules output by our blame analysis API enables them to find anomalies in query schedules or disproportionalities in resource allocations.

Discussion: While our current implementation is based on offline analysis after queries have completed execution, we show that this descriptive model still opens a huge space for contention exploration and blame attribution. An admin for a cluster scheduler can use the automated tips suggested by our extensible rule generator in managing her workload better. This feature is particularly useful for well-known or recurring executions. Our on-going research in designing an online contention analysis system, iQCAR-Live, aims to address the limitations discussed in this paper: that is, sensitivity of RATP to the availability of wait-time data, and application of rules automatically upon sensing interferences.

A BLAME FORMULATION FROM SECTION 4

We derive Equation (5) in Proposition 4.3 in this section to capture the blame $\beta^{r,h}_{ss\to tt}$ from a source stage to a target stage. *We assume that the resource r and host h are fixed, so we omit r, h in the superscripts and other places for simplicity where it is clear from the context.* First we discuss a simpler case, when there is a full overlap of tt with concurrently running tasks to present the main ideas. Then we discuss the general case with arbitrary overlap between st and tt.

(1) Full overlap of tt with concurrent tasks

Let the capacity of the host h to serve a resource r be $\mathcal{C}_h$ units/sec, *i.e.*, the minimum wait-time per unit of data processed for resource r on this host is $\mathsf{RATP}_h = \frac{1}{\mathcal{C}_h}$ sec/unit (see Definition 4.2). Therefore, it is also the best case (absence of any concurrency and contention) wait time for using resource r on host h. In any interval time τ to $\tau + \delta\tau$, total used capacity by all consumers of resource r is bounded by the system capacity $\mathcal{C}_h$. This can be expressed as:

$$\mathcal{C}_h = \mathcal{C}_{tt} + \mathcal{C}_1 + \mathcal{C}_2 + \cdots + \mathcal{C}_n + \mathcal{C}_u \tag{10}$$

where, $\mathcal{C}_{tt}$ is the capacity used by target task tt, $\mathcal{C}_1, \mathcal{C}_2, \dots\ \mathcal{C}_n$ are the individual used capacities by n concurrent tasks, and $\mathcal{C}_u$ represents the sum of capacities used by unknown causes and unused capacity. Rewriting Equation (10) using RATP values:

$$\frac{1}{\mathsf{RATP}^{\tau}_{h}} = \frac{1}{\mathsf{RATP}^{\tau}_{tt}} + \frac{1}{\mathsf{RATP}^{\tau}_{1}} + \frac{1}{\mathsf{RATP}^{\tau}_{2}} + \cdots + \frac{1}{\mathsf{RATP}^{\tau}_{n}} + \frac{1}{\mathsf{RATP}^{\tau}_{u}}$$

Multiplying by $\mathsf{RATP}^{\tau}_{tt}$ and subtracting 1 on both sides yields,

$$\frac{\mathsf{RATP}^{\tau}_{tt} - \mathsf{RATP}^{\tau}_{h}}{\mathsf{RATP}^{\tau}_{h}} = \frac{\mathsf{RATP}^{\tau}_{tt}}{\mathsf{RATP}^{\tau}_{1}} + \cdots + \frac{\mathsf{RATP}^{\tau}_{tt}}{\mathsf{RATP}^{\tau}_{n}} + \frac{\mathsf{RATP}^{\tau}_{tt}}{\mathsf{RATP}^{\tau}_{u}}$$

The left hand side above is the increase in RATP of target task and thus represents its slowdown $\mathcal{S}^{\tau}_{tt}$ given by (Definition 4.2)

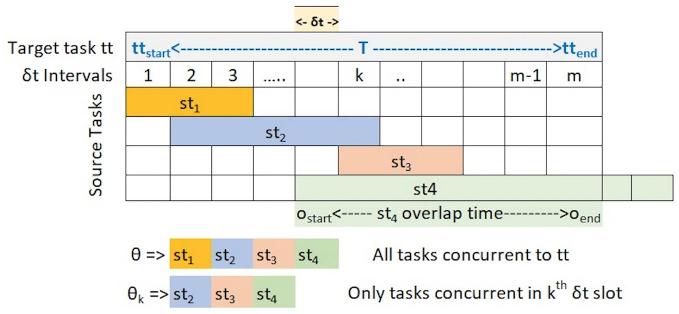
$$\mathcal{S}^{\tau}_{tt} = \left[\sum_{st\in n} \frac{\mathsf{RATP}^{\tau}_{tt}}{\mathsf{RATP}^{\tau}_{st}}\right] + \frac{\mathsf{RATP}^{\tau}_{tt}}{\mathsf{RATP}^{\tau}_{u}} \tag{11}$$

Each term on the right hand side of the above equation is contributed by one of the source tasks concurrent to task tt, and corresponds to blame attributable to a source task st in this interval, that is, $\beta^{\tau}_{st\to tt} = \frac{\mathsf{RATP}^{\tau}_{tt}}{\mathsf{RATP}^{\tau}_{st}}$.

(2) Partial overlap with concurrent tasks

The above equation works for a time interval in which all concurrent tasks have a total overlap with tt. In practice, they overlap for different length of intervals as can be seen from Figure 15.. As a result, we cannot use Equation (11) directly to attribute blame to a source task st. To derive blame for this scenario, we divide the total duration $T = tt_{end} - tt_{start}$ of task tt's execution time in $\delta\tau$ intervals such that in each $\delta\tau$ time-frame the above Equation (11) holds.

Let $\mathcal{S}_1, \mathcal{S}_2, \dots\ , \mathcal{S}_m$ be the slowdown in each of the $m = \frac{T}{\delta\tau}$ intervals of the task's execution. This gives us the mean slowdown of tt as (T_{tt} is the execution time of tt):



<- δt ->
Target task tt
tt_start<---------- T ---------->tt_end
δt Intervals
1 2 3 ..... k .. m-1 m
Source Tasks
st1
st2
st3
st4
o_start<----- st4 overlap time--------->o_end
θ => st1 st2 st3 st4
All tasks concurrent to tt
θk => st2 st3 st4
Only tasks concurrent in kth δt slot

Figure 15: An example overlap between concurrent tasks.

$$\mathcal{S}_{mean} = \frac{\delta\tau}{T_{tt}} \sum_{k \in m} \mathcal{S}_k$$

Substituing the value of slowdown $\mathcal{S}_k$ in the k_{th} interval using Equation (11),

$$\mathcal{S}_{mean} = \frac{\delta\tau}{T_{tt}} \sum_{k \in m} \sum_{st \in \theta_k} \frac{\mathsf{RATP}^k_{tt}}{\mathsf{RATP}^k_{st}}$$

where, θ_k is the set of source tasks that are concurrent in the k_{th} interval with task tt.
In order to understand the contribution of each source task towards the mean slowdown, the double summation can be rearranged as.

$$\mathcal{S}_{mean} = \sum_{st \in \theta} \sum_{k \in m'} \frac{\mathsf{RATP}^k_{tt}}{\mathsf{RATP}^k_{st}} \frac{\delta\tau}{T_{tt}}$$

where, for each st, m' is the number of intervals from time tt_{start} to tt_{end} in which st overlaps with tt. The outer sum is now on the set of all tasks, θ, that had an overlap with tt. The above equation in the limiting case is

$$\mathcal{S}_{mean} = \sum_{st \in \theta} \left[\int_{o^{start}_{st,tt}}^{o^{end}_{st,tt}} \frac{\mathsf{RATP}_{tt}}{\mathsf{RATP}_{st}} \frac{dt}{T_{tt}} \right]$$

where $o^{end}_{st,tt}$ - $o^{start}_{st,tt}$ is the overlap time between tasks tt and st. The integral inside the summation is the total blame towards a source task st, giving us the definition of blame in Equation (5). That is,

$$\beta_{st \to tt} = \frac{1}{T_{tt}} \left[\int_{o^{start}_{st,tt}}^{o^{end}_{st,tt}} \frac{\mathsf{RATP}_{tt}}{\mathsf{RATP}_{st}} dt \right] \qquad (12)$$

B ALGORITHMS

B.1 Impact Factor:

We compute IF_{uv} for an edge from node u at $Level_l$ to node v at $Level_{l-1}$ as the Vertex Contribution VC_u of u normalized by the total contribution of all parent nodes of v:

$$\mathsf{IF}_{uv} = \frac{\mathsf{VC}_u}{\sum_{w \in \mathsf{IN}(v)} (\mathsf{VC}_w)} \qquad (13)$$

Here $\mathsf{IN}(v)$ denotes the set of parent nodes w of v with an edge (w, v) in iQC-Graph. Therefore, for any node v, the sum of all IF_{uv} values is 1. Note that the IF_{uv} values for the edges (u, v) can be computed by a simple linear time graph traversal algorithm in $O(m + n)$ time starting with the topmost level, where m, n respectively denote the number of edges and nodes of the iQC-Graph. Algorithm 1 lists the steps involved in updating IF values of each edge in iQC-Graph.

Algorithm 1 Algorithm to update Impact Factors

```
1: Input: iQC-Graph(V,E)
2: Output: iQC-Graph(V,E) with updated IF
3: Set $\ell = 0$
4: Set $\ell_{max} = 5$
5: while $\ell \le \ell_{max}$ do
6:     curV$_\ell$ = Get all nodes at level $\ell$.
7:     for all $v \in$ curV$_\ell$ do
8:         $E_{in}$ = all edges incident on $v$
9:         $V_{in}$ = all source nodes of each edge in $E_{in}$
10:        Set cumulative_impact = $\sum_{j \in V_{in}} \left( VC^{\ell}_j \right)$
11:        for in_edge $\in E_{in}$ do
12:            in_node = source of in_edge
13:            in_node_impact= $VC_{in\_node}$
14:            $IF_{\text{in\_edge}} = \frac{\text{in\_node\_impact}}{\text{cumulative\_impact}}$        ▷ Definition 13
15:        end for
16:    end for
17:    $\ell = \ell + 1$
18: end while
```

Algorithm 2 Find top-k nodes with highest DOR at a level

```
1: Input: iQC-Graph($V$, $E$) output from Algorithm 1; $k$; $\ell_{in}$ (level of
   interest where top-$k$ explanations are sought)
2: Output: top-$k$ nodes in all levels in $\ell_{in}$
3: // Update Degree of Responsibility values for all nodes
4: Set $\ell = 1$ (for target stages) or = 0 (for target queries)
5: while $\ell \le \ell_{in}$ do
6:     curV$_\ell$ = Get all nodes at level $\ell$
7:     N$_{\ell+1}$ = all nodes in level $\ell + 1$ such that an edge $(u, v)$ exist to
   some $v \in$ curV$_\ell$
8:     for $u \in$ N$_{\ell+1}$ do
9:         Update DOR$_u$ using Definition 14
10:    end for
11:    $\ell = \ell + 1$
12: end while
13:
14: Return: $k$ nodes with highest DOR$_u$ at level $\ell_{in}$
```

Algorithm 3 Finding top-k impact paths in iQC-Graph

```
1: Input: iQC-Graph($V$, $E$) output by Algorithm 1; $k$
2: Output: top-$k$ DE with high impact
3: Initialize map_paths_to_impact = Empty
4: source_queries = Get all vertices at Level-6
5: for source ∈ source_queries do
6:     target_queries = Get all vertices at Level-0
7:     for target ∈ target_queries do
8:         all_paths = all paths from source to target
9:         for path ∈ all_paths do
10:            $u$ = source vertex of path, $v$ = target vertex of path
11:            path_weight = $\prod$ IF$_{u \to v}$
12:            Add path and path_weight to
13:                map_paths_to_impact
14:        end for
15:    end for
16: end for
17: sorted_paths = Sort map in map_paths_to_impact in decreasing val-
   ues (IF)
18: Return top-$k$ values from sorted_paths
```

Wait-Time Metric	Metric in IE level	Metric in DE level
Scheduling Waiting Time (SWT)	$\frac{SWT}{slots\_offered}$ where, $slots\_offered$ = number of slots offered to target task tt during its wait-time	$\frac{SWT_h}{slots\_offered_h}$
CPU Waiting Time (CWT)	$\frac{CWT}{input\_bytes}$ where, $input\_bytes$ = amount of data input for the task	$\frac{CPU\_THREAD\_BLOCKED\_ON\_MONITORS_h}{input\_bytes_h}$
		$\frac{CPU\_THREAD\_NOTIFICATION\_WAIT_h}{input\_bytes_h}$
Memory Waiting Time (MWT)	$\frac{MWT}{memory\_asked}$ where, $memory\_asked$ = total memory (storage or execution) asked by the task	$\frac{STORAGE\_MEM\_WAIT\_TIME_h}{storage\_memory\_asked_h}$
		$\frac{EXECUTION\_MEM\_WAIT\_TIME_h}{execution\_memory\_asked_h}$
Network Waiting Time (NWT)	$\frac{NWT}{remote\_bytes\_read}$ where, $remote\_bytes\_read$ = shuffle data read by target task tt	$\frac{NWT_h}{remote\_bytes\_read_h}$
IO Waiting time (IOWT)	$\frac{IOWT}{bytes\_processed}$ where, $bytes\_processed$ = total read or write bytes processed for the IO activity by the task	$\frac{IO\_READ\_WAIT\_TIME_h}{io\_bytes\_read_h}$
		$\frac{IO\_WRITE\_WAIT\_TIME_h}{io\_bytes\_written_h}$

Table 4: Resource Wait Time Metrics Split for Deep Explanations Level: in IE we store RATP values for only five metrics discussed in Section 2. For the DE level, in addition to maintaining these values per-host h, we split based on the resource in question. Here we list this breakdown of wait-time metrics capture at the DE level.

B.2 Degree of Responsiblity:

A Degree of Responsibility (DOR) of any vertex in iQC-Graph towards the contention faced by stage t-stage can be efficiently computed as:

$$\begin{aligned}\mathtt{DOR}_u &= \mathtt{VC}_v \ \textit{if the level of } u \textit{ is } 0\\ &= \sum_{v \in \mathtt{OUT}(v)} \mathtt{IF}_{uv} \times \mathtt{DOR}_v \ \textit{if the level of } u \textit{ is} \geq 1\end{aligned}$$

Here $\mathtt{OUT}(u)$ denotes the set of the child nodes w of u with an edge (u, w) in iQC-Graph. Intuitively, DOR gives the overall responsibility of any node toward the contention faced by the target query taking into account impacts of all its children with appropriate weights. Algorithm 2 elaborates on how we update the DOR values of each vertex in iQC-Graph and use those to find top-k explanations for every level.

Running time. Similar to IF, the values of DOR for the nodes of iQC-Graph can be computed by a linear $O(m + n)$ time algorithm. Now we process the graph in a bottom-up *topologically sorted order* (in contrast to a top-down traversal for computing IFs), where a node is processed only after all its children are processed.

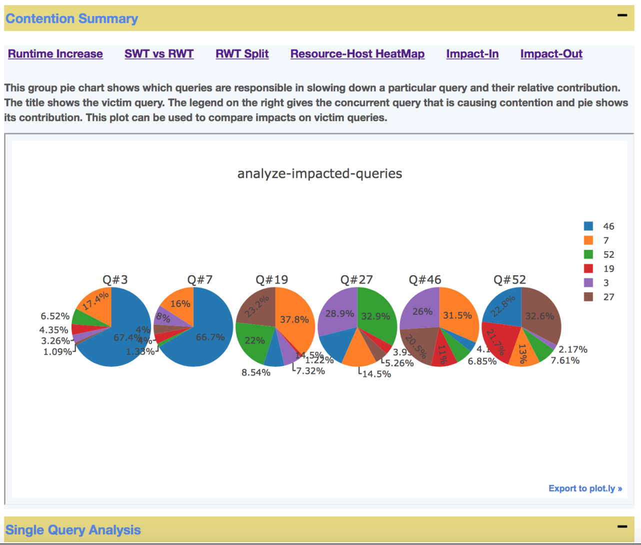
B.3 Finding top-k Impact Paths:

Finally, iQCAR enables users to identify a resource contention cause that potentially slows down multiple queries. Algorithm 3 gives the pseudocode for finding high impact paths in iQC-Graph.

Running time. Note that the number of paths in the graph is upper-bounded by $n_0 \times \Pi_{\ell=0}^{\ell_{max}-1} \mathtt{indeg}_\ell$, where n_0 is the number of nodes in Level 0, $\mathtt{indeg}_\ell$ is the maximum in-degree of nodes at Level ℓ from Level $\ell + 1$, and ℓ_{max} (= 6) is the highest level. As the structure of the expanations graph in Figure 4 shows, three of the levels have $\mathtt{indeg}_\ell = 1$, and even otherwise, the value is relatively small, therefore this algorithm runs efficiently despite looking at all possible paths in our graph.

C IQCAR UI VISUALIZATIONS

Figure 16 shows a sample screen from iQCAR UI showing an admin a summary of contentions faced by each target query. Figure 17 shows how the user can performa a detailed exploration for a single query.


Contention Summary

Runtime Increase | SWT vs RWT | RWT Split | Resource-Host HeatMap | Impact-In | Impact-Out

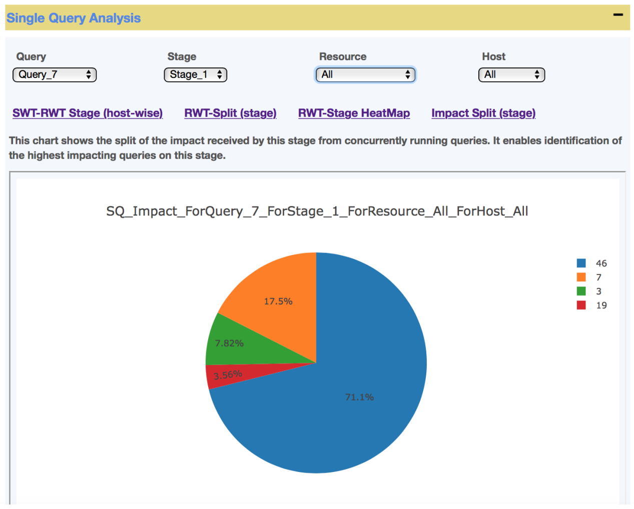
This group pie chart shows which queries are responsible in slowing down a particular query and their relative contribution. The title shows the victim query. The legend on the right gives the concurrent query that is causing contention and pie shows its contribution. This plot can be used to compare impacts on victim queries.

Figure 16: Sample visualizations from iQCAR UI shows a summary of contention impacts on each query. (b) shows an example of how users can drill-down to perform single query analysis.

REFERENCES

[1] [n. d.]. Apache Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. ([n. d.]).

[2] [n. d.]. Netflix and Stolen Time. https://www.sciencelogic.com/blog/netflix-steals-time-in-the-cloud-and-from-users. ([n. d.]).



This chart shows the split of the impact received by this stage from concurrently running queries. It enables identification of the highest impacting queries on this stage.

Figure 17: Sample visualizations from `iQCAR` **UI shows an example of how users can drill-down to perform single query analysis.**

[3] [n. d.]. Teradata. http://www.teradata.com. ([n. d.]).
[4] [n. d.]. The Noisy Neighbor Problem. https://www.liquidweb.com/blog/why-aws-is-bad-for-small-organizations-and-users/. ([n. d.]).
[5] [n. d.]. Vertica. https://www.vertica.com. ([n. d.]).
[6] 2016. Apache Ambari. http://ambari.apache.org. (2016). [Online; accessed 01-Nov-2016].
[7] 2016. Apache Parquet. https://parquet.apache.org. (2016). [Online; accessed 01-Nov-2016].
[8] 2016. Ganglia Monitoring System. http://ganglia.info. (2016). [Online; accessed 01-Nov-2016].
[9] 2016. Spark Monitoring and Instrumentation. http://spark.apache.org/docs/latest/monitoring.html. (2016). [Online; accessed 01-Nov-2016].
[10] 2016. TPC Benchmark™DS . http://www.tpc.org/tpcds/. (2016). [Online; accessed 01-Nov-2016].
[11] Mumtaz Ahmad, Ashraf Aboulnaga, and Shivnath Babu. 2009. Query interactions in database workloads. In *Proceedings of the Second International Workshop on Testing Database Systems*. ACM, 11.
[12] 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. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. ACM, 1383–1394.
[13] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory*. 316–330.
[14] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.
[15] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.
[16] Doug Cutting. 2016. Apache Hadoop. http://hadoop.apache.org. (2016). [Online; accessed 01-Nov-2016].
[17] Anupam Datta, Deepak Garg, Dilsun Kaynar, Divya Sharma, and Arunesh Sinha. 2015. Program actions as actual causes: A building block for accountability. In *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 261–275.
[18] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle.. In *CIDR*. 84–94.
[19] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 59–72.
[20] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. 2012. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 4.
[21] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: towards automated SLOs for enterprise clusters. In *Proceedings of OSDIâĂŹ16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 117.
[22] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. 2012. Perfxplain: debugging mapreduce job performance. *PVLDB* 5, 7 (2012), 598–609.
[23] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. 2010. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *PVLDB* 4, 1 (2010), 34–45.
[24] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 34 (2016), 1–7.
[25] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout
[26] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. 2015. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1167–1178.
[27] Sudeepa Roy and Dan Suciu. 2014. A formal approach to finding explanations for database queries. In *SIGMOD*. 1579–1590.
[28] spark-sql-perf team. 2016. Spark SQL Performance. https://github.com/databricks/spark-sql-perf. (2016). [Online; accessed 01-Nov-2016].
[29] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *PVLDB* 2, 2 (2009), 1626–1629.
[30] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.
[31] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *PVLDB* 6, 8 (2013).
[32] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.
[33] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1599–1614. https://doi.org/10.1145/2882903.2915218
[34] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 265–278.
[35] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113
[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*.
[37] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 379–391.