iQCAR: inter-Query Contention Analyzer for Data Analytics Frameworks

Prajakta Kalmegh
Duke University
Durham, North Carolina
pkalmegh@cs.duke.edu

Shivnath Babu
Unravel Data Systems
Palo Alto, California
shivnath@unraveldata.com

Sudeepa Roy
Duke University
Durham, North Carolina
sudeepa@cs.duke.edu

ABSTRACT

Resource interferences caused by concurrent queries is one of the key reasons for unpredictable performance and missed workload SLAs in cluster computing systems. Analyzing these inter-query resource interactions is critical in order to answer time-sensitive questions like 'who is creating resource conflicts to my query'. More importantly, diagnosing whether the resource blocked times of a 'victim' query are caused by other queries or some other external factor can help the database administrator narrow down the many possibilities of query performance degradation. We introduce iQCAR, an inter-Query Contention Analyzer, that attributes blame for the slowdown of a query to concurrent queries. iQCAR models the resource conflicts using a multi-level directed acyclic graph that can help administrators compare impacts from concurrent queries, identify most contentious queries, resources and hosts in an online execution for a selected time window. Our experiments using TPCDS queries on Apache Spark show that our approach is substantially more accurate than other methods based on overlap time between concurrent queries.

KEYWORDS

Performance evaluation; contention analysis; blame attribution; resource interference; data analytics frameworks

ACM Reference Format:
Prajakta Kalmegh, Shivnath Babu, and Sudeepa Roy. 2019. iQCAR: inter-Query Contention Analyzer for Data Analytics Frameworks. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319904

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06...$15.00
https://doi.org/10.1145/3299869.3319904

1 INTRODUCTION

In today's data-driven world, there is a growing demand of autonomous data processing systems [10]. One of the critical roadblocks in achieving the desired goal of automation is ensuring predictable query performance in multi-tenant systems. The question – "*Why is my query slow?*" – is nontrivial to answer in standard big data processing systems employing shared clusters. The authors have seen firsthand how enterprises use an army of support staff to solve problem tickets filed by end users whose queries are not performing as they expect. An end user can usually troubleshoot causes of slow performance that arise from her query (e.g., when the query did not use the appropriate index, data skew, change in execution plans, etc.). However, often the primary cause of a poorly-performing query is low-level resource contentions caused by other concurrently executing queries in a multi-tenant system [5, 9, 14, 16, 32]. For example, in one of our experiments, one query was found to be 178% slower than its unconstrained execution due to resource conflicts. Diagnosing such causes of unpredictable performance is difficult and time consuming requiring in-depth expertise of the system and the workload. Today, cluster administrators have to manually traverse through intricate cycles of query interactions to identify how interferences on resources affect desired performances of concurrently running queries.

Should the solution be prevention or diagnosis (and cure)? To ensure a predictable query performance, preventive measures often provide query execution isolation at the resource allocation level. For example, an admin tries to reduce conflicts by partitioning resources among tenants using capped capacities [2], reserving shares of the cluster [18], or dynamically regulating offers to queries based on the configured scheduling policies like max-min fair [30] or 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. The allocations are primarily based on only a subset of the resources leaving the requirements for other shared resources unaccounted for. An approach solely based on preventive techniques will also have other limitations since real-life

```
SELECT i.i_brand_id, sum(ss_ext_sales_price) sum_agg
FROM   date_dim dt, store_sales ss, item i
WHERE  dt.d_date_sk = ss.ss_sold_date_sk
AND ss.ss_item_sk = i.i_item_sk AND i.i_manufact_id = 128
GROUP  BY i.i_brand_id;
```

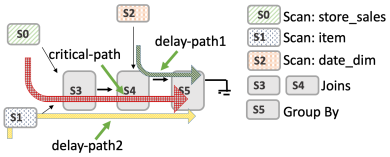


Figure 1: Execution DAG of TPCDS Query 3 showing the computation that each stage performs). Stages S0, S1 and S2 are IO intensive as they scan input data. S3, S4 are network, IO intensive owing to shuffle operation required for Join. S5 is more CPU bound due to the aggregate operation.

workloads are a mix of very diverse types of queries. Therefore, low-level resource conflicts continue to impact queries in shared clusters, thus inviting a need to supplement prevention techniques with techniques for diagnosis of contentions such that required actions can be taken.

Our research focuses on the latter. In this paper, we present iQCAR - a tool to detect resource contentions between concurrent queries using *blocked times* (time a task is blocked for a resource) [24]. While there have been several attempts to diagnose root causes (systemic, configuration, external or plan-related) for the slowdown of a query [17, 20, 23, 29], to the best of our knowledge iQCAR is the first attempt to answer an important question - whether and how the *blocked times* of a query are affected by low-level conflicts caused by other concurrent queries.

1.1 Data analytics frameworks

This paper focuses on query/job execution on data analytics frameworks like MapReduce [19] and Spark [31]. These frameworks are designed to perform complex logic on large distributed datasets by parallelizing computations. Every query is broken down into a DAG of stages where each stage accomplishes a particular piece of overall logic on the input or intermediate data. Figure 1 shows an example of a query that is broken into six stages. A stage consists of multiple parallel tasks, where the tasks are the actual execution units that perform the same computation on different blocks of the input data. Since these blocks are distributed across the cluster, tasks of a stage execute in parallel on different hosts and their output is exchanged in a shuffle operation with tasks of dependent stages.

In these frameworks, tasks are executed using pipelining to enable parallel use of CPU, disk and network. They execute in a single thread and use an iterator model to read, process and output each record through the pipeline. As disk and network requests are handled in the background by OS, a task can use and wait for multiple resources at the same time. It can however be completely blocked only when the specific resource required for the execution of next line of code is not available. Figure 2 shows the execution timeline of a single task. It uses network and disk for reading data, CPU for processing and again disk for writing the results. The execution logic of this task is to first read and process data from remote machine and then process local disk data. The program flow of the task is the following: a network read request is issued at time t_0 and the task is blocked for availability of first block of network data until t_1. Task execution continues until t_2 when it is blocked for CPU allocation by the OS. The task resumes and continues execution until t_3 on previously retrieved network data (even though no new data is received from network in this duration). At t_3, the task blocks again for more remote data until t_4 when data is ready. The task blocks again at t_5 for OS CPU allocation and completes processing remote data by t_6. At t_6, it issues request for disk IO read and is blocked until t_7 when some data is available. The task continues its execution without any further blocking despite more wait time for disk IO. This is because it has enough data to process without being blocked at every instance of its remaining execution.

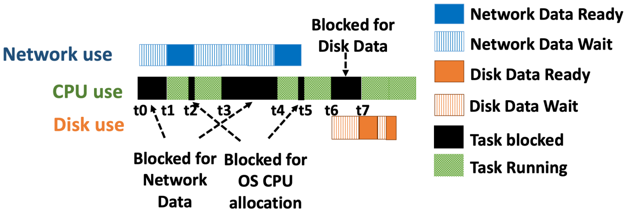


Figure 2: The execution timeline of a single task. Task execution is blocked for different resources during its execution and does not always synchronize with resource wait times.

BlockedTime: During its execution, a task can wait multiple times for network and disk data (refer to *Disk Data Wait* and *Network Data Wait* in Figure 2). However, the sum of these wait times does not add up to its total blocked time (refer to *Task Blocked* in Figure 2). This is because waiting for a resource does not imply that a task is unable to make progress. The effective impact of any concurrent task/process on the slowdown of a task is therefore only to the extent to which it increases the blocked time of the task. In [24], authors demonstrated the role of using the time tasks are blocked on disk IO and network for effective performance analysis of data analytical workloads; in iQCAR, we use blocked times of tasks on disk IO, network, CPU and memory as the basis to calculate the metrics of slowdown and blame attribution.

1.2 Challenges in Contention Analysis

Consider the dataflow-DAG of a data analytical TPCDS Query 3 (referred to as Q_0 henceforth) shown in Figure 1. Suppose

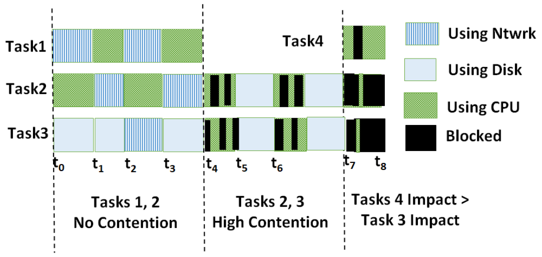


Figure 3: Example overlap between four tasks.

an admin notices a slowdown for Q_0 compared to a previous execution using some automated tool or manual analysis - then we designate Q_0 as our victim query. As a first step of troubleshooting, she wants to identify whether Q_0 was a victim of concurrency-caused contention or not (i.e., whether the reasons were systemic or configuration instead). If yes, which of the concurrent queries, say, Q_1 or Q_2 is more responsible for the slowdown of Q_0.

The common approach adopted today toward addressing such questions may involve different steps: (1) using historical executions, identify which stages of the query slowed it down. (2) for each stage, use low-level monitoring tools (e.g., [8]) to identify the intervals or hosts with unusual activity, and find resources responsible for its bottlenecks. (3) finally, find the overlapping queries/stages/tasks and analyze their resource utilization further to diagnose whether and why they caused the bottlenecks. We call this approach of blame attribution as *Deep-Overlap*. This process is time-consuming and error-prone since it requires expertise of the involved systems, and an in-depth understanding of the workload. In particular, an administrator faces the following challenges.

Challenge 1. Analyzing Contentions on Dataflows: A query can slowdown due to delay in one or more of its component stages. Some delays propagate to the end while others get mitigated by faster later stages. Figure 1 has two such paths, *delay-path1* and *delay-path2* that get mitigated, but the *critical-path* contributes to the final slowdown. Identifying and accounting for paths of highest impact is important and challenging for contention analysis[1].

Challenge 2. Analyzing Multi-Resource Contentions: In data analytics frameworks, tasks interleave their resource usages due to the pipelining model of execution (see Section 1.1). The consumption and contention for any resource is non-uniform and depends on the mix of concurrent tasks. We identify two issues that arise from this:

Challenge 2a: *Deep-Overlap* can be misleading: As an example, consider the tasks in Figure 3. Though all three tasks ($Task1$, $Task2$ and $Task3$) execute in parallel between time t_0 to t_4, tasks $Task2$ and $Task3$ do not cause any contention to each other due to no overlap for any resource. On the other hand, they have complete overlap for both resources (CPU and disk IO) between time t_4 and t_7, thereby facing high contention.

Challenge 2b: Quantifying blame for contentions is hard: Since tasks contend for multiple resources simultaneously, a task may get one resource faster but other resource slower than a competing task. Even when tasks are contending for single resource, some tasks may cause greater slowdown than others. In Figure 3, both $Task3$ and $Task4$ overlap for same resource with $Task2$ between t_7 and t_8. The impact caused by $Task4$ is however higher as it gets the highest share. In fact, $Task3$ and $Task2$ impact each other equally. Identifying such complex interactions is difficult but necessary to accurately capture and quantify contentions.

1.3 Our Contributions

We have built the system iQCAR with a goal to address the above challenges. The explanations generated by iQCAR can help an admin understand why a query is slow in an execution, or isolate the most contentious queries that use the same resources or nodes in the cluster, which might take hours of effort without the help of iQCAR. We present the system architecture of iQCAR in Section 2, and make the following contributions:

- **Blame Attribution:** We use the Blocked Times [24] values for multiple resources (CPU, Network, IO, Memory[2]) to develop a metric called *Resource Acquire Time Penalty* (RATP) that aids us in computing *blame* towards a concurrent task while addressing Challenge 2 **(Section 3)**.
- **Explanations and blame analysis:** We present a multi-level Directed Acyclic Graph (DAG), called iQC-Graph, that enables distribution of blame at different granularity. We generate explanations for resource conflicts faced by a query by traversing this graph **(Section 4)**.
- **End-to-end system:** We have instrumented Apache Spark [31] to collect the time-series data on the blocked time and resource usage metrics for tasks. Our web-based front-end [22] allows users to get workload-level contention summary plots, or perform step-wise exploration of impacts using iQC-Graph. We discuss our implementation and the current limitations of iQCAR system **(Section 5)**.
- **Experimental evaluations:** We evaluated iQCAR using various test-cases conducted on TPCDS workloads running on Apache Spark. We also compare iQCAR with two alternative approaches and demonstrate its better accuracy compared to them **(Section 6)**.

[1] The number of paths is typically very high for analytical queries that involve many joins and aggregations.

[2] It is not the physical memory but application memory cache managed by frameworks like [31] to dynamically trade between storage of intermediate data and execution requirements [4].

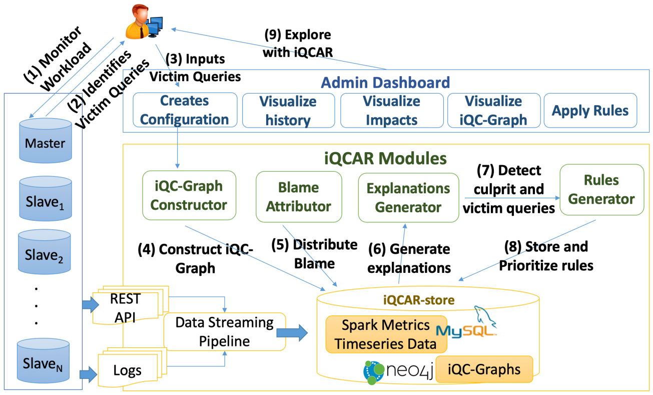


Figure 4: iQCAR **System Architecture**

Section 7 discusses the related work and we conclude in **Section 8** with directions for future research. Some details are moved to the appendix due to space constraints.

2 SYSTEM OVERVIEW OF IQCAR

The iQCAR system enables users to detect contentions online while the queries are executing, and perform a deep exploration of contentious scenarios in the cluster offline. Figure 4 shows the architecture of iQCAR. In Step (1), an admin uses the available user interface (UI) to identify a set of queries to be analyzed. Each of the queries chosen for deep exploration in Step (2) is termed as a **victim query**, its stages as **victim stages**, and its tasks as **victim tasks**. Once the user submits victim queries to iQCAR in Step (3), the **Graph Constructor** builds a multi-level DAG (Section 4) for these queries in Step (4). In addition, users can configure the resources or hosts for which they want to analyze the contentions. For example, users can diagnose the impact of concurrency on only CPU contention on all or a subset of hosts, or originating from potential culprit queries submitted by a particular user, etc. The **Blame Attributor** module (Section 3) then computes and distributes blame to all vertices in the graph in Step (5), which are then used to update the edge weights of the graph subsequently. The edge weights are then used by the **Explanations Generator** module to generate explanations (defined in Section 4.2) of one query impacting another in Step (6). The edge weights are also used to update a *degree of responsibility* (DOR) metric for each node in the graph to assign relative impact values and enable a ranking. Finally, the potential queries (creating contention) with their scores produced by the Explanation Generator module are examined by the admin to understand the contention in the system in Step (7) [3].

3 BLAME ATTRIBUTION

In this section we explain the key concept in iQCAR: how it attributes blame to concurrent queries. An important aspect of iQCAR is that it does not rely on any information of previous executions. A query is considered to be running slow if its execution is blocked (see Blocked Time in Section 1.1) because some other concurrent query is using the same resource. A concurrent query is blamed for slowdown if it consumes resources at a higher rate than the victim query[4]. If more than one concurrent query is responsible for blocking then their blame values are in the ratio of their resource consumption rates. Specifically, a concurrent query acquires blame for the slowdown of a victim query if its tasks (1) execute on the same machine as the victim tasks, (2) have overlapping run times with the victim tasks, and (3) consume the same resources at a higher rate than the victim tasks.

3.1 Resource Acquire Time Penalty (RATP)

First we define a measure called RATP to denote the time spent by a task to acquire one unit of resource (e.g., CPU, Disk, Network, etc.) on a host. For example, $\text{RATP}_{network}$ is the time spent fetching a single record (or byte) of data from the network.

Definition 3.1. *For a given resource r and a host h, suppose a victim **task** vt consumes δr units of resource in a small time interval δt seconds. Then the **Resource Acquire Time Penalty** (*RATP*) for task vt for resource r on host h in the interval δt is*

$$RATP^{\delta t}_{vt,r,h} = \frac{\delta t}{\delta r} \tag{1}$$

In the remaining section, we assume that the resource r, host h and the interval δt are fixed unless mentioned otherwise, so we omit r, h in the subscripts and δt from superscripts for simplicity where it is clear from the context. This simplifies Equation (1) for a task vt to

$$RATP_{vt} = \frac{\delta t}{\delta r} \text{sec / unit resource} \tag{2}$$

3.2 Slowdown of a Task

Let the capacity of host h to serve resource r be C unit resource/sec. The *minimum time* to acquire one unit of r on this host can be expressed as (omitting subscript h):

$$\text{RATP}^{*} = \frac{1}{C} \text{ sec / unit resource} \tag{3}$$

[3] Our system also includes a prototype of a basic *Rule Generator* module that suggests heuristics to avoid contentions (like alternate query placement, dynamic priority readjustment for stages and queries, etc.). However, building a sophisticated rule generator is a focus of our current research and is out of scope of this paper. Instead, here we focus on the topic of contention analysis and detection of contention creating queries.

[4] This assumes that both queries have equal right on the resource. It is easy to incorporate weights/priorities into this definition by normalizing the consumption rates by weights/priorities.

Definition 3.2. *The total* ***slowdown*** *of task vt in time interval δt due to unavailability of resource r is defined as:*

$$\mathcal{S}_{vt} = \frac{(\mathsf{RATP}_{vt} - \mathsf{RATP}^*)}{\mathsf{RATP}^*} \tag{4}$$

where RATP^* *is the capacity of host h for resource r (see (3)).*

Example 3.3. *Consider three tasks Task1, Task2 and Task3 reading* 30, 60 *and* 120 *bytes of data from disk in 1 second on a machine with* 210 *bytes/sec IO speed. In this single time interval, vt = Task1 is thus slowed by* 6 *times the ideal rate ($\mathcal{S}_{vt} = \frac{210}{30} - 1 = 6$ from (4)).*

Intuitively, the slowdown captures the deviation from the ideal resource acquisition rate on the host h and gives a measure of the excess delay incurred for unit resource in the δt execution interval. The slowdown of vt will be zero when it has the entire resource to itself. Thus, $\mathcal{S}_{vt}$ is the slowdown caused by all other running processes in the system. We classify them into 3 categories:

(1) concurrently running tasks,
(2) known external processes (e.g., framework processes common to all tasks), and
(3) unknown external processes (e.g., processes not known in advance or not managed).

Thus the slowdown is expressed as:

$$\mathcal{S}_{vt} = \underbrace{(\sum_{ct=1}^{n} \beta_{ct\rightarrow vt})}_{p_1} + \underbrace{(\sum_{i=1}^{M} \beta_{known,i\rightarrow vt}) + \beta_{unknown\rightarrow vt}}_{p_2} \tag{5}$$

Here $\beta_{ct\rightarrow vt}$ is the blame assigned to each of the n tasks $ct = 1, \cdots, n$ concurrently running with vt; $\beta_{known,i\rightarrow vt}$ is the blame assigned to other known $i = 1, \cdots, M$ non-conflict-related causes that contribute to the wait time of vt. $\beta_{unknown\rightarrow vt}$ captures the blame attributable to unknown factors.

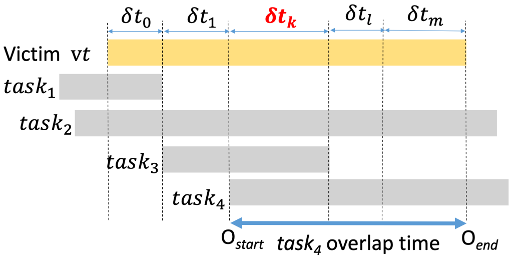
3.3 Blame with RATPs

The first term, p_1, in equation (5) is a sum of the blame values of n concurrent tasks of a victim task. The **blame** $\beta_{ct\rightarrow vt}$ for the contention caused for resource r by a concurrent task ct to a victim task vt on host h can be expressed as equation (6). How equation (6) can be derived from equations (4) and (5) is given in Appendix A and B, and we outline the intuition and illustrate with examples below:

$$\beta_{ct\rightarrow vt} = \left[\sum_{\delta t \in \mathcal{O}} \frac{\mathsf{RATP}_{vt}^{\delta t}}{\mathsf{RATP}_{ct}^{\delta t}}\right] \tag{6}$$

Here $\mathcal{O}$ is the set of δt time intervals in which tasks ct and vt overlap, and we omit the subscripts r and h. Figure 5 shows an example overlap of four concurrent tasks with vt in $m + 1$ intervals of its execution.

Example 3.4. *In Example 3.3, the blame value for Task2 is* 60/30 = 2 *and blame for Task3 is* 120/30 = 4 *using (6).*



All tasks concurrent to vt: $task_1$, $task_2$, $task_3$, $task_4$
Only Tasks concurrent in **k**th interval: $task_2$, $task_3$, $task_4$

Figure 5: An example overlap between concurrent tasks.

The above blame formulation is based on RATP values which are by definition resource specific (we are omitting subscripts r for simplicity). A concurrent task can only be blamed if it uses the same resource at a higher rate (addresses Challenge 2a). Since blame is also a ratio of RATP values, the value of blame is thus more for tasks consuming at higher rate with respect to the victim task (addresses Challenge 2b). In the next subsection, we give a more accurate computation of blame using blocked times.

3.4 Incorporating Blocked Time in Blame

As discussed in Section 1.1, in data analytics frameworks tasks can continue to execute even when they wait for some of the resources. In such scenarios, the blame computed using equation (6) can be inaccurate. Suppose in Examples 3.3 and 3.4, while $Task1$ waits $\frac{210-30}{210} = \frac{6}{7}$ sec within the $\delta t = 1$ sec interval, it is only blocked for $\frac{1}{4}$ sec (in the rest of the time it is still running). The absence of concurrent tasks $Task2$ and $Task3$ can only speed it up by $\frac{1}{4}$ sec at most and hence they only deserve to be blamed to that extent. Therefore using blocked time for calculation of RATP of victim task gives a more accurate blame value. In another scenario, say the tasks also contend for CPU in the same 1sec and receive CPU time slots in the same proportion. If we sum the blame for all resources in an interval, then it will double although there might be a overlap in wait times for resources (CPU and IO). This can again be mitigated if we ensure that there is no overlap between wait times used in blame calculation. The solution again is to use blocked times. As by definition, the blocked time of a task is from the view point of its computation progress, it is only counted once even if it is due to wait on more than one resource. We now update the definition of RATP from Equation 2 using blocked time as:

$$\mathsf{RATP}\text{-}blocked_{vt} = \frac{\mathsf{BT}_{vt}}{\delta r_{vt}} \tag{7}$$

where BT_{vt} = the blocked time of the victim task vt when it consumed δr_{vt} units of resource in interval δt. Using this definition, the blame value in Equation 6 is re-written as:

$$\hat{\beta}_{ct \to vt} = \sum_{\delta t \in m} \frac{\mathsf{RATP}\text{-}\mathit{blocked}_{vt}}{\mathsf{RATP}_{ct}} \tag{8}$$

Any concurrent task (irrespective of the amount of its resource consumption) is considered blocking if it consumes the same resource on which the impacted task is blocked. However, in this new formulation, the blame attributed is more for tasks that consume more resource. As a result, while all concurrent tasks block each other, the impact from a concurrent task with the highest resource share is highest.

The RATP value of a concurrent task *ct* (denominator) is still based on the entire time interval for two reasons: First, it ensures that the slowdown based on blocked time has an upper bound as derived in Equation 5. The numerator on every term on the right hand side (RHS) decreases (as blocked time in any interval strictly bound by it) but the denominators have no change. If we were to change the RATP of concurrent tasks also to be based on blocked time then terms on the RHS could either increase or decrease which does not guarantee any bound. Second, for the other known and unknown processes entities in $p2$ term in Equation 5, it is easier to obtain the resource consumed in an interval by any external process in comparison to its blocked time.

4 GLOBAL BLAME DISTRIBUTION

In the previous section, we discussed our methodology to assign blame to a *concurrent task ct* for causing contention to a victim task vt. As discussed in Section 1.1, a query in data analytics frameworks is processed by multiple stages that have many parallel tasks. iQCAR uses a multi-layered directed acyclic graph to capture, aggregate, and compute contentions between queries at different granularity and dimensions (i.e., stage level, resource level and also host level). The different levels in our graph-based model are chosen carefully to address the challenges discussed in Section 1.2.

4.1 iQC-Graph

Our graph model consists of seven levels designed to (i) drill down from a query to its tasks for every resource and host, (b) assign blame to concurrent tasks, and finally (c) aggregate blame to concurrent stages and queries. The vertices are constructed bottom-up from Level-0 to Level-6. For each node u in the graph, we assign weights, called *Blame Contributions* (denoted by BC_u) that are used later for analyzing impact and generating explanations. The BC values are computed for Level-3 first using Equation 9 (discussed shortly), and are then updated middle-out for all other levels. This blame value represents accountability towards the blocked time faced by the victim query, and is distributed to all nodes in the graph. Thus, the unit of BC for each node at every level is in *seconds*. Intuitively, it represents the fraction of the total blocked time in *seconds* faced by a victim query that is attributed to that node. A detailed construction of the graph is explained in Appendix C. Thus for all levels, BC_u measures the blame assigned to u for causing slowdown to a single victim query vertex in Level-0. Figure 6 illustrates the distribution of blame for all levels in an example iQC-Graph. Level-5 shows the stages of only concurrent queries. There are, however, other causes that can cause contention like external known processes and unknown processes (see Section 5.1), which are not broken down into stages. In Figure 6, we therefore short-circuit their impact from Level-4 to Level-6 (notice no vertices at Level-5 for External-IO and Unknown).

Level-0 to 3 - Tracking Blocked Times at Different Granularity: The BC of vertices from Level-0 to 3 represent their contribution towards the delay faced by Q_i node at Level-0. For a node u in each level, BC_u gives the blocked time for the entity represented at that level.

- $\mathsf{BC}_u^{\ell_3}$ (Level-3): represents the cumulative blocked time for all tasks of a victim stage node u for resource r on host h. It is the lowest level of granularity in the iQC-Graph.
- $\mathsf{BC}_u^{\ell_2}$ (Level-2): The values at Level-3 are aggregated per resource to capture resource level blocked times in Level-2, i.e., $\mathsf{BC}_u^{\ell_2} = \sum_{h \in hosts} \mathsf{BC}_h^{\ell_3}$. In data analytics frameworks, the computation done by each task is completely independent of other tasks in the same stage. The tasks can all run in parallel or sequentially one at a time depending on the cluster workload and scheduling situation. To make the logic of iQC-Graph independent of task parallelism, the blocked times are aggregated to reflect the total potential improvement if there was no blocked time. Moreover, since the unit of blame attributed is in *seconds*, these impacts on tasks of the same stage executing across different hosts can be aggregated at Level-2. The invariant in equation (12) is valid per host/machine. Hence, the blame values are assigned to concurrent tasks executing on same machine, and thus clock synchronization is not required when aggregating the blame values.
- $\mathsf{BC}_u^{\ell_1}$ (Level-1): The blame for victim stages at Level-1 is the aggregate value of blocked times due to individual resources i.e., $\mathsf{BC}_u^{\ell_1} = \sum_{r \in resources} \mathsf{BC}_r^{\ell_2}$.
- $\mathsf{BC}_u^{\ell_0}$ (Level-0): $\mathsf{BC}_u^{\ell_0} = \sum_{vs \in vic\_stages} \mathsf{BC}_r^{\ell_1}$. A query DAG can consist of multiple parallel paths (see Challenge 1), the blocked time of a query cannot be computed by summing up the blocked time of all its stages. To address this concern, we consider only the stages on the **critical path** of a query's executionas its victim stages. These are the sequence of stages that form the longest chain of execution for Q_i (sum of run times of stages on the critical path gives the total runtime of the query). In our example in Figure 1, stages $s_0 \to s_3 \to s_4 \to s_5$ form the *critical path* of Q_0.

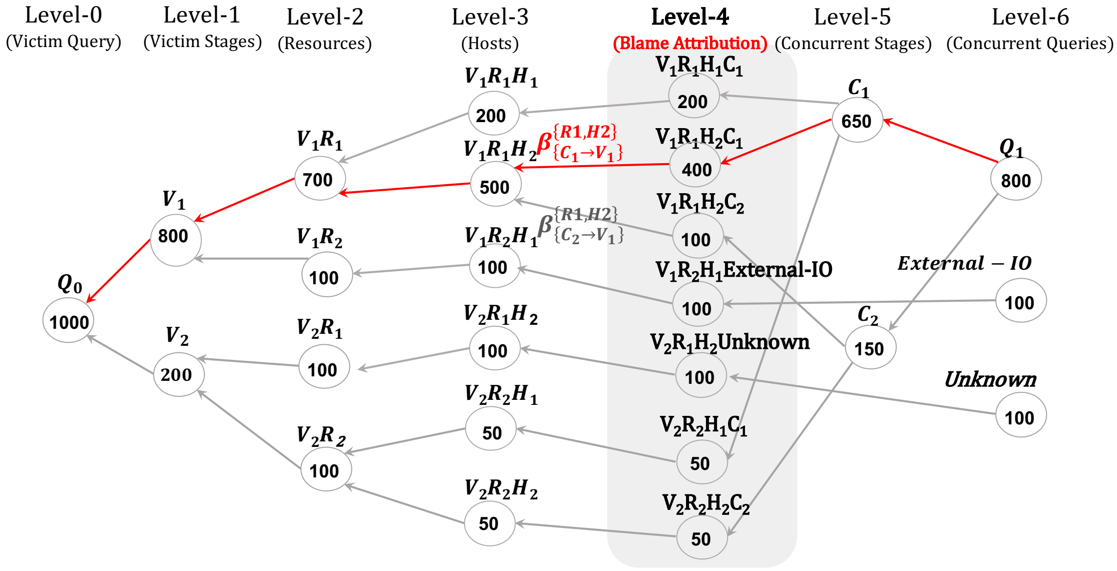


Figure 6: An example iQC-Graph. The labels above nodes represent the entity ID, and the values in the vertices give the BC. The highlighted β values show an example of how Equation 8 is used to distribute blame (blocked-time) from Level-3 vertex to Level-4 vertices. The unit of blame contribution of each node in iQC-Graph is in *seconds*. The path in red shows the highest-impact path. Figure shows how iQCAR attributes blame to other external apps causing, say, IO impact. All unaccounted blocked-time is attributed to *Unknown* source.

Level-4 - Linking Cause to Effect: For every victim stage, we begin by identifying all tasks concurrent to tasks of this stage and compute their blames using Equation 8 for every r, h combination. The blame for a single concurrent stage C_j of another query is then computed by aggregating individual blames assigned to its tasks ct, and the victim tasks vt:

$$\hat{\beta}_{cs\to vs} = \sum_{ct,vt} \hat{\beta}^{r,h}_{ct\to vt} \tag{9}$$

The $\mathsf{BC}^{\ell_3}_u$ at Level-3 for a $u = (vs, r, h)$ is then distributed among Level-4 nodes (we add m nodes P_u in Level-4 for m concurrent stages, see Appendix C) in proportion to their blames values. That is,

$$\mathsf{BC}^{\ell_4}_u = \frac{\hat{\beta}_{cs\to vs}}{\sum_{cs'\in P_u} \hat{\beta}_{cs'\to vs}} * \mathsf{BC}^{\ell_3}_u \tag{10}$$

Intuitively, it gives the fraction of the total blocked-time on host h for resource r (Level-3 node) which is attributed to C_j. For example, if $\mathsf{BC}^{\ell_3}_u = 500sec$, and the ratio of blames from $C_1 : C_2$ are $4 : 1$ respectively (from above equation), then $\mathsf{BC}^{\ell_4}_{V1,R1,H2,C1} = 400$ and $\mathsf{BC}^{\ell_4}_{V1,R1,H2,C2} = 100$ (see Figure 6).

Level-5 and 6 - Aggregating Blame: After we compute BC values at Level-4, we track the sources of their incoming edges (concurrent stages). For each outgoing edge $\langle u, v\rangle$ from Level-5 to all Level-4 vertices corresponding to a single victim query, the value of $\mathsf{BC}^{\ell_5}_u = \sum_{v\in edge\_targets} \mathsf{BC}^{\ell_4}_v$. Similarly, we compute $\mathsf{BC}^{\ell_6}_u = \sum_{v\in edge\_targets} \mathsf{BC}^{\ell_5}_v$. These BCs give the total impact originating from this source (stage or query) toward a single victim query. For multiple victim queries at Level-0, we maintain a map of BC values originating from each node at Levels 5 and 6 towards each victim.

4.2 Explanations and their Scores

While the BC values are sufficient to answer the question "*who* is slowing me down?" for a particular victim query, we cannot use this measure *as-is* to compare the impacts caused or received by queries. For example, suppose Q_3 causes an impact of $500sec$ to each of Q_1 and Q_2. It is possible that this impact was just 1% of the total impact received by Q_1, whereas it was 100% of the impact received by Q_2. The responsibility of Q_3 toward the slowdown of each query is thus different. We thus cannot use the blame value of $1000sec$ originating from it towards two victims to rank its outgoing impact. To address this, the Explanations Generator module uses the BC values to compute two measures, namely the **Impact Factor** (IF) and the **Degree of Responsibility** (DOR) that together provide a normalized basis for comparing impacts. We set the IF as edge weights and DOR as the node properties.

Impact Factor (IF): From Level-0 to 4, for every edge $\langle u, v\rangle$ in iQC-Graph, its Impact Factor IF_{uv} is the normalized impact received by each child node v from its parent nodes u-s. For instance, the impact from a Level-3 node u to a Level-2 node v is $\mathsf{IF}_{uv} = \frac{\mathsf{BC}^{\ell_3}_v}{\mathsf{BC}^{\ell_2}_u}$. Figure 7 shows an example of the

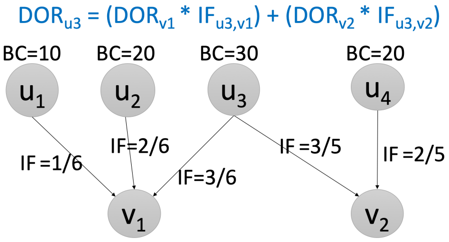


Figure 7: Example computation of IF and DOR from BC.

impact received by child node v_1 from its parents u_1, u_2, u_3. For edges $\langle u, v\rangle$ from Level-5 to 4, $\mathtt{IF}_{uv} = \frac{\mathtt{BC}_v^{\ell_4}}{\mathtt{BC}_u^{\ell_5}}$, and from Level-6 to 5, $\mathtt{IF}_{uv} = \frac{\mathtt{BC}_v^{\ell_5}}{\mathtt{BC}_u^{\ell_6}}$.

Degree of Responsibility (DOR): iQCAR consolidates the IF further to aggregate responsibility of each entity (queries, stages, resources and hosts) towards the contention faced by every victim query. The value of $\mathtt{DOR}_u$ is in the range $[0, 1]$ and is computed as the *sum of the weights of all paths from any node u to the victim query node t, where the weight of a path is the product of all* $\mathtt{IF}_{vw}$ *values of all the edges* (v, w) *on this path.* However, the DOR values can be efficiently computed in linear time by graph traversal as illustrated in Figure 7 for node v_3. 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.

Candidate Explanations: A path in iQC-Graph starting from a culprit query cq and culminating at a victim query vq, thus, represents a candidate explanation for the contention caused by cq to vq. We represent it as follows: $\phi = Expl($ vq, vs, res, res′, host, cs, cq, $\mathcal{P})$
where ,
vq and vs denote the victim query and stage being explained by ϕ resp;
res $\in CPU, Memory, IO, Network$;
res' is the type of resource impacted (see Section 5.2);
host is the host of impact;
cs is the impacting stage of the concurrent query;
cq is the impacting concurrent query;
$\mathcal{P}$ is the cumulative weight of the path originating from cq and ending at vq. Users can rank the explanations using $\mathcal{P}$ to filter top paths of contentions from cq Level-6 to its victim.

We show various use-cases in which iQCAR uses the candidate explanations and their DOR scores in Appendix E.

5 IMPLEMENTATION

In this section, we discuss some specific details of our implementation of iQCAR on Apache Spark [31]. They include the supported resources, instrumentation in the framework, frequency of metrics collection and our approach to handle the terms in $p2$ in Equation 5.

5.1 Impact of Non-Query Processes

In Section 3.2 we separated the causes for query slowdown into three categories. While iQCAR is designed to accurately handle the blame due to the first category (concurrent queries) it can also be used to identify contentions due the second category (other known processes) and classify remaining blame to final category.

Known Processes: iQCAR is pre-configured for attributing blame to select known causes i.e., in every interval iQCAR identifies the resource consumption of these processes/threads to calculate their blame. Its API allows users with domain knowledge to configure support for more external processes. For instance, consider the Java garbage collector (GC) process. As Spark runs on JVM its tasks are subject to GC pauses. We model this as a long running concurrent query GC_Q with one task on every host. The metrics collector captures the time spent on GC in every interval of blame calculation and attributes the appropriate blame to this task. If a query is running slow because of high garbage collection activity then iQCAR identifies GC_Q as the cause.

External Processes We model two constant long running queries *External-IO*, *External-Network* with one task each on every host. We keep a track of the total system resource usage for disk and network during each execution window. This information is used to compute the resource consumed by external processes (by subtracting the aggregate resource usage for all tasks from the total system usage). The *External-IO* and *External-Network* synthetic queries are then attributed blame using these computed values.

5.2 Supported Resources

Our implementation supports contention analysis for all four system resources network, disk, cpu and memory. We collect system-level metrics by deploying per-host agents that capture and report with every heartbeat. For query level metrics, we use existing Spark metrics where available and our instrumentation for additional metrics as outlined below:

- Network: Existing *fetch-wait-time* and *shuffle-bytes-read* are used.
- Memory: Spark manages application memory by splitting them into execution and storage memory buckets. Instrumentation is used to capture *storage memory wait time*, *storage memory acquired*, *execution memory wait time* and *execution memory acquired* metrics.
- IO: Existing *scan-time* and *bytes read* metrics are used for disk read. For disk write, *shuffle-write-time* and *shuffle-bytes-written* metrics are used.
- CPU: This contention has two components (a) *Lock and Block wait* due to contention for common data-structures (locks, monitors, etc). Java JMX based instrumentation is used to capture these wait time metrics (b) *OS scheduling*

wait owing to usage being 100% (as more tasks need CPU than max runnable cores). It is computed by subtracting all above blocked times from the interval wall time [5].

5.3 Frequency of Metrics Collection

For each task vt, we added support to collect the time-series data for its blocked-time and the corresponding data processed metrics at the boundaries of task start and finish for all other tasks ct concurrent to vt. Figure 5 shows the four cases of task start end boundaries concurrent to task vt. Note that with this approach, the length of intermediate δt depends on the frequency of arrival and exit of concurrent tasks, thus enabling us to capture the effects of concurrency on the execution of a task more accurately. For workloads consisting of tasks with sub-second latency, our approach gives fine-grained windows for analysis. However, if the arrival rate of concurrent tasks is low (long-running tasks), this can affect the distributions of our metric values. To address this, we also record the metrics at heartbeat intervals in addition to above task entry and exit boundaries. Section 6.3.1 compares the impact of both the approaches on the quality of our explanations.

5.4 Limitations

Concurrently executing queries can cause impacts in many indirect ways too. The indirect impacts are more profound when queries share common framework and/or process resources like process managed shared memory, shared cache etc. As an example, in our environment (Spark SQL over Thriftserver), tasks of multiple queries run in the same JVM process thereby having a high heap memory coupling. If a task related to one query puts stress on the heap memory then the resultant garbage collection pause impacts all other tasks. In iQCAR we handle this specific issue by creating a separate GC task in the list of our known causes as discussed previously to avoid incorrect blame attribution but still fall short of accurately pinning the blame to problematic task. Another challenge arises when the impact from concurrent queries is not negative. In some cases they may aide faster processing. Incorporating such indirect slowdowns and accounting for positive vs negative impacts is part of our on-going effort.

5.5 Discussion

While our focus in this paper is on SQL workloads on Spark, iQCAR's approach of (a) using blocked times, (b) its blame attribution model and (c) DAG based blame propagation generalize well to other workloads on any data-flow based processing system. Spark was our choice of implementation owing to its ability to process different workloads like SQL, machine learning, graph analytics, etc. within a common framework. All the existing metrics (except *scan-time*) and those from our instrumentation are in its core engine and could be used for any workload. iQCAR could be adapted to work with any other similar system by implementing metrics collection module for that system. The complexity of this task depends on existing support (metrics) from the system and instrumentation effort for missing metrics. In Appendix F we describe the implementation of our metrics collector module in another sql-on-hdfs system Presto [12].

6 EXPERIMENTAL EVALUATION

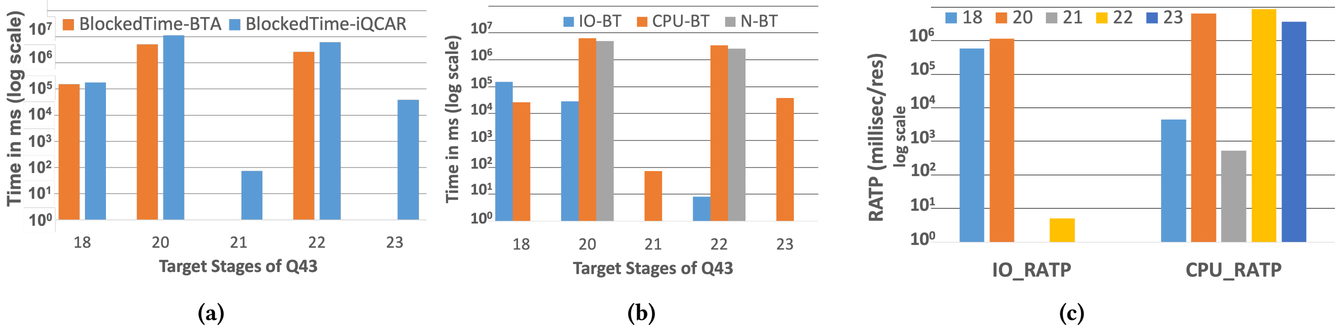
Our experiments were conducted on Apache Spark 2.2 [31] deployed on a 20-node local cluster (master and 19 slaves). Spark was setup to run using the standalone scheduler in fair scheduling mode [30] with default configurations. Each machine in the cluster has 8 cores, 16GB RAM, and 1 TB storage. A 300 GB TPCDS [15] dataset was stored in HDFS and accessed through Spark SQL in Parquet [3] format. The SQL queries were taken from [27] without any modifications.

Workload: Our analysis uses a TPCDS benchmark workload that models multiple users running a mix of data analytical queries in parallel. We have 6 users (or tenants) submitting their queries to dedicated queues. Each user runs 15 sequential queries randomly chosen from the TPCDS query set. The query schedules were serialized and re-used for all experiments to compare results across executions. We identify a victim query as the one that took most hit (suffered maximum slowdown) compared to its unconstrained execution (when run in isolation). The query Q_{43}, which was 178% slower, is the victim discussed in the rest of this section.

6.1 Debugging Challenges Without iQCAR

The purpose of this experiment is to show how iQCAR enables deeper diagnosis of contentions compared to other approaches. For comparison with baseline, we use the following metrics: (a) **Blocked-Time Analysis (BTA)**: blocked times for IO and Network [24] aggregated at stage and query levels, (b) **Naive-Overlap**: based only on the overall query overlap times (a technique popularly used by support staff trying to resolve *who is affecting my query* tickets), and (c) **Deep-Overlap**: we compute the cumulative overlap time between all tasks of concurrent queries. In this approach, overlap time of tasks executing in parallel is aggregated in comparison to previous one where only the maximum overlap is considered. For both overlap-based approaches, highest blame is attributed to query with most overlap.

[5] It is commonly recommended to run more task threads than CPU threads [25] for increased CPU utilization; but, this also leads to contention in some intervals. It is a trade off that users make based on experience. Our implementation is on default values



(a)

(b)

(c)

Figure 8: (a) Cumulative blocked times from iQCAR and BTA approach; (b) Blocked times for IO, CPU and Network; (c) Comparison of RATP for IO and CPU for top-5 impacted stages of Q_{43}.

Figure 8a compares the blocked times observed (y-axis is plotted on log-scale) with the *BTA* method vs captured by iQCAR for its top-5 victim stages. Since iQCAR accrues blocked times for additional supported resources (discussed in Section 5.2), it gives more insight into the slowdown for stages like 21 and 23. Figure 8b compares the relative per resource blocked times. However, to identify disproportionality in these blocked-times (i.e. understanding impact) RATP is better. Blocked-time gives the possible speed-up only if there are no resource constraints, i.e., infinite resource supply, it is still insufficient to provide relative impact in a real cluster with resource constraints. Here it is difficult for the admin to infer whether stage 20 or 22 caused more impact due to concurrency. Based on cumulative-blocked times alone, stage 20 stands out as cause for query slowdown. However, when we compare the RATPs for these stages using iQCAR, we see in Figure 8c that the CPU RATP was much higher for multiple stages compared to their IO RATP, whereas the network RATP was significantly low to even compare. If we analyze the impact on each of these stages generated using the explanations module of iQCAR, query Q_{43} received highest impact from victim stage 22 through CPU.

Clearly, Block Time helps in identifying the parts of a query whose speed-up would help the most but not necessarily the parts that are hit due to contention. In a resource constrained environment it is the latter that is more helpful for administrators to align at-least the query scheduling strategies.

We now compare the results of different overlap time approaches for two stages 20 and 22. An admin, based on higher blocked times, would infer that queries concurrent (or the ones with highest overlap) to these stages have caused highest impact to Q_{43}. However, as shown in Figure 9, the top overlapping or concurrent queries differ significantly between *Naive-Overlap* and *Deep-Overlap*. The queries with minimal overlap shown in *Naive-Overlap* (Q_4 and Q_{27}) have relatively more tasks executing in parallel on the same host as that of victim query, causing higher cumulative *Deep-Overlap*. Clearly, using *Naive-Overlap* can lead to misleading results for blame attribution.

The output from iQCAR is more comparable to *Deep-Overlap*, but has different contributions. Especially for Q_{11}, where the tasks had a high overlap with tasks of our victim query Q_{43}, the impact paths showed low path weights between these end points. A further exploration revealed that only 18% of the execution windows (captured via the time-series metrics), showed increments in CPU and IO acquired values for Q_{11} in the matching overlapping windows. Q_{11} itself was blocked for these resources in 64% of the overlapping windows. Without the impact analysis API of iQCAR, an admin will need significant effort to unravel this and is more subject to falsely attribute blame to either Q_{52} with *Naive-Overlap* or to Q_{11} with *Deep-Overlap*.

6.2 Test Cases

While it is important to diagnose and distribute accurate blames, it is also required in a timely manner for any actionable measures. The purpose of these experiments is to demonstrate how iQCAR can perform a time-series analysis to detect culprit queries induced in an online workload. For ease of exposition, we restrict our definition of culprit to the most impacting query for a single victim. For each test-case, the culprit query was formulated to create a specific resource

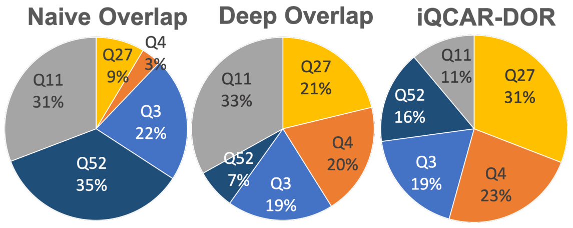
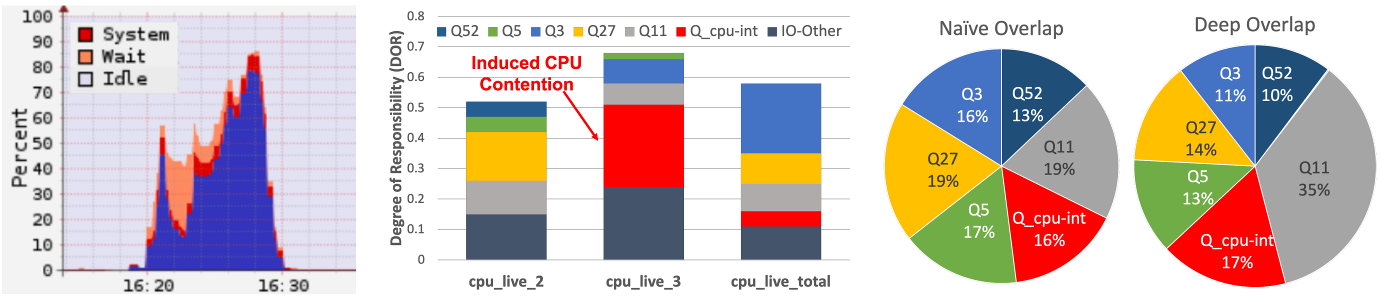


Figure 9: Compare top-5 impacting queries between (a) *Naive-Overlap* (b) *Deep-Overlap* and (c) iQCAR.



(a) (b) (c)

Figure 10: *CPU-Internal* experiment: (a) Ganglia snapshot showing higher CPU Utilization when $Q_{cpu-int}$ was running; (b) DOR of $Q_{cpu-int}$ is about 27% towards query Q_{43} during CPU induction; (c) Overlap-based impact from concurrent queries.

Table 1: Summary of Induced Contention Scenarios

Test-Case	Detail
CPU-Internal	run CPU-intensive query *after* Q_{43} starts.
IO-External	read a large file on every host *after* Q_{43}.
Mem-Internal	Cache *web_sales* table *before* Q_{43} starts.

contention scenario as summarized in Table 1. Note that, (a) test-case of *Memory-external* is out of scope since iQCAR detects contentions only for managed memory in Spark, (b) inducing only network contention in our SQL workload was not possible as our attempts to increase shuffle data resulted in an increase IO contention owing to heavy shuffle write.

6.2.1 CPU-Internal. Listing 1 shows our induced culprit query $Q_{cpu-int}$:

Listing 1: $Q_{cpu-int}$: *CPU-Internal* Induction query

```
with temp (select c1.c_first_name as first_name,
sum(sha2(repeat(c1.c_last_name,45000),512)) as ssum
from customer c1, customer c2
where c1.c_customer_sk = c2.c_customer_sk
group by c1.c_first_name)
select max(ssum) from temp limit 100;
```

We first discuss the characteristics of $Q_{cpu-int}$: (a) low-overhead of IO read owing to a scan over two small TPCDS *customer* tables each containing 5 million records stored in parquet format, (b) low network overhead since we project only two columns, (c) minimal data skew between partitions as the join is on *c_customer_sk* column which is a unique sequential id for every row and shuffle operation used hash-partitioning, and (d) high CPU requirements owing to the *sha2* function on a long string (generated by using the repeat function on a string column). Figure 10a shows that the CPU utilization reaches almost 80% during our induction.

Observations: iQCAR is used to analyze the contentions in different time windows of the workload execution (here *cpu_live_1*, *cpu_live_2*, *cpu_live_3*, and *cpu_live_total*). Figure 10b shows the change in DOR values of the concurrent queries towards Q_{43}. As it starts execution only in the second window, we skip *cpu_live_1* in the figure. The stacked bars show the relative contribution from each of the top-5 culprit queries, and their heights denote the total contribution from them. $Q_{cpu-int}$ was induced at the end of *cpu_live_2*, hence no contribution from $Q_{cpu-int}$ in this window. As seen in *cpu_live_3*, once we induce $Q_{cpu-int}$ when Q_{43} starts, iQCAR correctly detects its contribution of 27%.

The end-of-window analysis in *cpu_live_total* shows the overall impact to Q_{43} from all concurrent queries during its end-to-end execution. Although $Q_{cpu-int}$ caused high contention to Q_{43} for a period, its overall impact was still limited (0.05%). Without iQCAR, if the admin performs *Naive-Overlap* and *Deep-Overlap* in *cpu_live_3* window to attribute blame as shown in Figure 10c, she will be wrongly attributing about 35% of received impact to Q_{11}, whereas, the actual impact shows less than 10% overall impact from Q_{11}.

6.2.2 IO-External. In this test-case, we show how iQCAR can be used to detect impact from culprit processes that run outside the Spark framework. To create an IO-intensive external culprit process, we read and dump a large file on every host in the cluster after the workload stabilizes (at least one query is completed for each user). Let us call this culprit process as $IO-Other$. The timing of induction was chosen such that it overlaps with the scan stage of Q_{43}. We created a 60GB file on each host and used the command in Listing 2 to read in blocks of size 256MB using the below command:

Listing 2: $IO-Other$: *IO-External* Induction query

```
dd if=~/file_60GB of=/dev/null bs=256
```

Observations: Ganglia showed over 1600% aggregate disk utilization for all nodes (19 slaves) in the cluster during this period of IO induction. We analyze impacts for the following windows: (a) Q_{43} had not started in **io_live_1**, (b) **io_live_2**

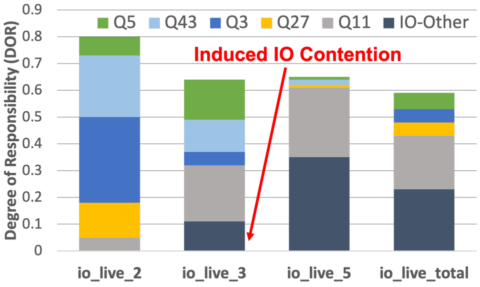


Figure 11: DOR of culprit query $IO-Other$ **towards query** Q_{43} **changes as the workload progresses.**

analysis was done for victim query Q_{43} just before we induced $IO-Other$, (c) **io_live_3** to **io_live_5** windows while $IO-Other$ is running concurrently with Q_{43} (we omit showing **io_live_4** due to plotting space constraints), (d) Q_{43} finishes execution before **io_live_6**, and (e) the **io_live_total** window to analyze overall impact on Q_{43} from the beginning of the workload till the end. Figure 11 shows the relative contributions from each of the concurrent queries, showing that iQCAR detects the culprit process first in **io_live_3** (shown in dark blue), and outputs an increasing impact during **io_live_5** when it peaks.

6.2.3 Mem-Internal. In this third test-case we show how iQCAR distinguishes between multi-resource interferences and rightly detects a culprit that impacts a single resource largely. To achieve this, we create a memory contention scenario which should potentially also cause heavy IO conflict. Since we monitor contention only for the managed memory within Spark, our internal memory-contention test-case caches a large TPCDS table (*web_sales*) in memory just before Q_{43} is submitted. Let's call this query as $Q_{mem-int}$. Note, with alternate approaches like CPU Stolen Time [9], this induced culprit causing memory conflict will go undetected.

Observations: We now analyze the impact on Q_{43} for the following four windows: (a) **mem_live_2** is the period where both Q_{43} and $Q_{mem-int}$ had begun execution together (Q_{43} had not started in **mem_live_1**), (b) **mem_live_3** shows a window where some other queries had entered the workload, and both Q_{43} and $Q_{mem-int}$ were still running, (c) **mem_live_4** and **mem_live_5** are the windows when Q_{43} was still running and $Q_{mem-int}$ had finished execution, and (d) **mem_live_total** analyzes the overall impact on Q_{43} from top-5 culprit queries during its complete execution window. While Q_{43} was running concurrently with multiple other queries, $Q_{mem-int}$ causes more than 40% share of the total impact received via memory once it begins as shown in Figure 12. Note that Q_{43} scans *store* and *store_sales*, whereas we cached *web_sales* in our induction query. This created high IO interference, but the value of blame distributed via memory was noted to be highest.

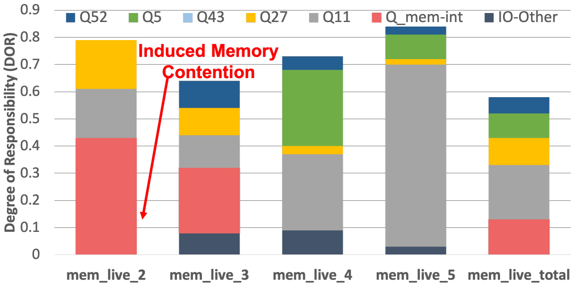


Figure 12: Impact of induced memory contention on Q_{43}**.**

6.3 Instrumentation

In this section, we analyze the impact of our instrumentation on the quality of explanations and query runtimes.

6.3.1 Frequency of Data Collection. iQCAR aggregates resource usage and blocked times metrics from small intervals to calculate blame. The explanations (i.e., DOR values) are derived from these blame values and are sensitive to the interval size. In this experiment, we show that the accuracy of explanation improves with decrease in interval length. In Section 5.3 we describe how a combination of regular interval and task event based metrics gives most accurate blame (TE) and use it as ideal value to calculate DOR deviation. Figure 13 shows that the DOR values vary for different heartbeat intervals and their average deviation (vector distance) from ideal (shown on secondary y-axis as 'DOR distance from Ideal (TE)' and depicted in red) tends to improve with decreasing intervals. While lower interval lengths give higher accuracy the ideal interval size depends on average task runtimes of the workload. In our workload (TPCDS), task runtimes vary highly so we used a lower value (2s) to ensure better accuracy. We next discuss the overheads associated with this instrumentation.

6.3.2 Instrumentation Overhead. Many metrics relevant to iQCAR are already provided by frameworks (e.g., Spark (Section 5.2) and Presto (Section F)) and instrumentation for additional metrics is very low (book-keeping instructions to capture time and bytes used). Although the absolute overhead increases with frequency of metrics collection for larger tasks it is still a small fraction of their runtime. The metrics are also collected asynchronously to avoid tasks from blocking. Figure 14 shows the overhead (collection interval of 2s) over baseline (collection interval 10s) as the query concurrency increases. Increase in concurrency increases the average runtime of tasks due to resource contentions. The

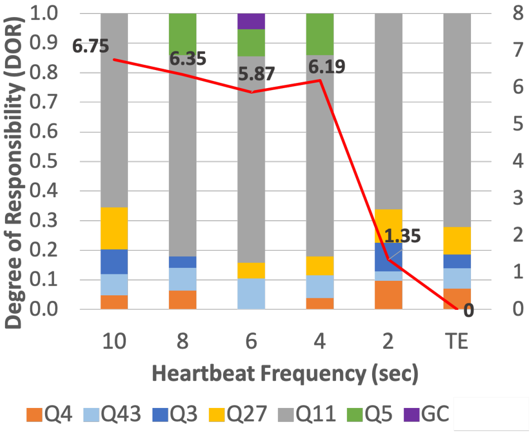


Figure 13: Impact of varying intervals of metrics collection on explanation DORs. ***TE*** **denotes the metrics collection at task-event boundaries (Section 5.3) in addition to 2s logging.**

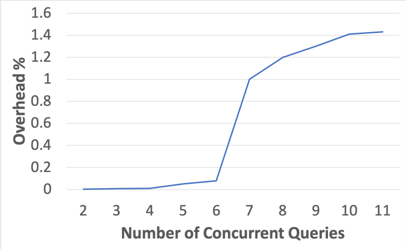


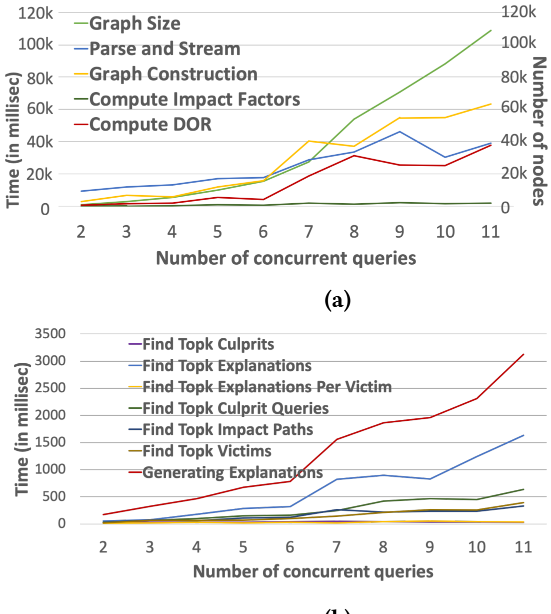
Figure 14: Instrumentation in Spark - Percentage overhead over baseline.

overhead increases only by 1.4% for 5 fold increase in concurrency.

6.4 Analysis

In this section we discuss the performance of analytic components of iQCAR for entire workload (offline) and describe the changes in online analysis.

6.4.1 Graph Construction. In Figure 15a, we observe (a) an increase in size of iQC-Graph and (b) time taken for execution of different components, as the number of concurrent queries increases. The window of analysis here is the entire runtime of workload and the graph is built for every query. For example, for 6 concurrent queries, iQC-Graph was built with six vertices at Level-0, all their stages in Level-1 and so on. The size of the graph therefore grows very quickly as the concurrency increases. This is shown on the right y-axis. The graph size increases from under 100 for 2 queries to about 120k for 11 queries. The time for constructing the graph is shown on the left y-axis. Even for a single threaded execution it only grows at half the rate of graph size. It increases from under 2s for 2 queries to 60s for 11 queries. This is still under 2.5% of the runtime of the workload. The computation time of IF and DORs is still lower, under 1% of the workload time. In a live scenario the window of analysis is much smaller and hence the size of the iQC-Graph reduces significantly. The nodes in Levels 2, 3, and 4 corresponds to a particular victim stage, the subgraph formed by these nodes for one victim stage at Level 1 is disjoint from the subgraph formed by the nodes for another victim stage. They are connected back at Level 5 if multiple victim stages execute concurrently with the same culprit stage. This property allows the construction of subgraphs from Level 0 to Level 4 in parallel.



(a)

(b)

Figure 15: (a) Graph Construction - The time taken by the Graph Constructor module in various steps. (b) Explanations Generation - The time taken by the Explanations Generator module to invoke its Blame Analysis API for different use-cases.

6.4.2 Explanations Generation. In Figure 15b, we present the times spent for different types of analysis algorithms. The process of generating textual explanations from the contention iQC-Graph (the plot-line for 'Generating Explanations' depicted in *red*) dominates the runtime of our *Blame Analysis* process. The y-axis values for this measure account for the time to generate explanations for all the use-cases (a top-*k* query) listed in the figure. However, in a live-contention analysis scenario, a user will be performing a single use-case at a time. Moreover, since our API uses customized Cypher [6] queries on Neo4j to retrieve data and generates plots in real-time using Plotly [11], users can interactively explore contentions.

Table 2: Comparison of iQCAR with other approaches

(Category) Related Work	No Historical data	Detects slowdown	Detects bottlenecks	Blame attribution	Dataflow Aware
(1) Ganglia, Spark UI, Ambari	✓		✓		
(2) Starfish, Dr. Elephant, OtterTune		✓	✓		
(3) PerfXplain, Blocked Time, PerfAugur		✓	✓		
(3) Oracle ADDM, DIADS	✓		✓		✓
(3) DBSherlock		✓			✓
(4) CPI^2	✓	✓	✓^{CPU}	✓	
iQCAR	✓	✓	✓	✓	✓

7 RELATED WORK

iQCAR is designed to analyze inter-query resource contentions in near real-time. It does not aim to understand impact of configuration changes and does not use data of any previous execution. While the contention analysis logic of iQCAR is not dependent on previous data, identifying specific queries (among entire workload) to analyze will be easier with some reference information. In this paper, we use the unconstrained execution (without any interference) time of a query for this purpose. In practice, as mentioned in Section 1.2, an admin may identify victim queries using other performance criteria (e.g., SLA, missed-deadline etc.). or by simply looking for query with maximum blocked time. To the best of our knowledge, there is no system today that performs inter-query contention analysis on data analytics frameworks without any data from previous executions. We compare our work with other approaches below and give a summary in Table 2.

(1) Monitoring Tools: Cluster monitoring tools like Ganglia [8] and application tools like Spark UI [13] and Ambari [1] provide query metrics at a high level. They do not capture low-level resource interactions.

(2) Configuration recommendation: Tools like Starfish [21], Dr.Elephant [7], and OtterTune [28] analyze performance and suggest changes in configuration. However, it is difficult to predict how these system-wide changes will affect inter-query interactions in an online workload.

(3) Root Cause Diagnosis tools: Performance diagnosis has been studied in the database community [17, 20, 29], for cluster computing frameworks [23, 24], and in cloud based services [26]. While the design of iQCAR is motivated by some concepts from such prior work, the techniques and goals differ as follows: **(a) Database Community:** In **ADDM** [20], *Database Time* of a SQL query is used for impact analysis. iQCAR instead takes an approach to provide an end-to-end contention analysis while also enabling deep exploration of contention symptoms. DIADS [17] uses Annotated Plan Graphs that combine the details of database operations and Storage Area Networks (SANs) to provide an integrated diagnosis tool. The problem addressed in DIADS is not related, but our multi-level explanation framework bears similarity to their multi-level analysis. Causality based monitoring tools like **DBSherlock** [29] diagnose problems using data from previous executions. **(b) Cluster Computing: PerfXplain** is a debugging toolkit that uses a decision-tree approach to provide explanations for the performance of MapReduce jobs. Unlike iQCAR, it also depends on previous executions. **Blocked Time** metric [24] emphasizes the need for using resource blocked times for performance analysis of data analytical workloads; we critically use blocked time but do a finer analysis to identify the role of concurrent queries in causing these blocked tmes. **(c) Cloud: PerfAugur** [26] detects anomalous system behavior and generates detailed explanations for them, whereas iQCAR generates explanations for the slowdown due to resource conflicts.

(4) Detecting Antagonist Queries: CPI^2 [32] uses Cycles-Per-Instruction data from hardware counters to identify antagonist queries but is limited to CPU contention.

8 CONCLUSION

Resource interferences due to concurrent executions are one of the primary and yet highly misdiagnosed causes of query slowdowns in shared clusters today. This paper discusses some of thechallenges in detecting accurate causes of contentions, and illustrateswhy blame attribution using existing methodologies can be inaccurate. We propose a theory for quantifying blame for slowdown, and present techniques to filter genuine concurrency related slowdowns from other known and unknown issues. We further showed how our graph-based framework allows for consolidation of blame and generate explanations allowing an admin to explore the contentions and contributors of these contentions systematically. An interesting direction of future research is to develop a contention-aware cluster scheduler that can dynamically reprioritize contentious or victim queries, and/or delay stage submissions to avoid possible resource conflicts.

ACKNOWLEDGMENTS

We are thankful to our anonymous reviewers for their valuable feedback that helped us improve the paper. This work was supported in part by NSF awards IIS-1408846, IIS-1423124, IIS-1552538, IIS-1703431 and NIH Award 1R01EB025021-01.

REFERENCES

[1] 2019. Apache Ambari. http://ambari.apache.org.

[2] 2019. Apache Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[3] 2019. Apache Parquet. https://parquet.apache.org.

[4] 2019. Apache Spark: Memory Management Overview . https://spark.apache.org/docs/latest/tuning.html#memory-management-overview.

[5] 2019. Collection of small tips in further analyzing your hadoop cluster. https://www.slideshare.net/Hadoop_Summit/t-325p210-cnoguchi.

[6] 2019. Cypher Query Language. https://neo4j.com/developer/cypher.

[7] 2019. Dr. Elephant. http://www.teradata.com.

[8] 2019. Ganglia Monitoring System. http://ganglia.info.

[9] 2019. Netflix and Stolen Time. https://www.sciencelogic.com/blog/netflix-steals-time-in-the-cloud-and-from-users.

[10] 2019. Oracle Autonomous Database. https://www.oracle.com/database/autonomous-database/index.html.

[11] 2019. Plotly: Modern Visualization for the Data Era. https://plot.ly.

[12] 2019. Presto: Distributed SQL Query Engine for Big Data. https://prestodb.github.io.

[13] 2019. Spark Monitoring and Instrumentation. http://spark.apache.org/docs/latest/monitoring.html.

[14] 2019. The Noisy Neighbor Problem. https://www.liquidweb.com/blog/why-aws-is-bad-for-small-organizations-and-users/.

[15] 2019. TPC Benchmark™DS . http://www.tpc.org/tpcds/.

[16] 2019. Understanding AWS stolen CPU and how it affects your apps. https://www.datadoghq.com/blog/understanding-aws-stolen-cpu-and-how-it-affects-your-apps/.

[17] Nedyalko Borisov, Shivnath Babu, Sandeep Uttamchandani, Ramani Routray, and Aameek Singh. 2009. Why Did My Query Slow Down? *arXiv preprint arXiv:0907.3183* (2009).

[18] 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.

[19] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[20] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle.. In *CIDR*. 84–94.

[21] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: a self-tuning system for big data analytics.. In *Cidr*, Vol. 11. 261–272.

[22] Prajakta Kalmegh, Harrison Lundberg, Frederick Xu, Shivnath Babu, and Sudeepa Roy. 2018. iqcar: A demonstration of an inter-query contention analyzer for cluster computing frameworks. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1721–1724.

[23] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. 2012. Perfxplain: debugging mapreduce job performance. *PVLDB* 5, 7 (2012), 598–609.

[24] 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

[25] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. 2013. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *Proceedings of the Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2013)*.

[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] spark-sql-perf team. 2016. Spark SQL Performance. https://github.com/databricks/spark-sql-perf. [Online; accessed 01-Nov-2016].

[28] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1009–1024.

[29] 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

[30] 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.

[31] 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

[32] 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.

A SLOWDOWN OF A TASK FROM CONCURRENCY

In this section, we derive Equation 6 from Equation 4. For ease of presentation, we repeat some details as we intend to present the entire derivation with a single continuity.

Consider a victim task vt that wants to consume resource r on host h. In the δt interval, let the capacity of the host h to serve the resource r be $\mathcal{C}$ unit resource/sec. The *minimum time* to acquire one unit of r on host h can be expressed as:

$$\mathsf{RATP}^* = \frac{1}{\mathcal{C}} \text{ sec / unit resource} \tag{11}$$

The total capacity $\mathcal{C}$ (of a resource) is consumed by all the processes running on the system. These processes include (a) tasks related to queries, (b) known processes (e.g.common framework services) (refer Section 3) and (c) other unknown system processes. This can be expressed as

$$\mathcal{C} = \mathcal{C}_{vt} + \mathcal{C}_1 + \mathcal{C}_2 + \cdots + \mathcal{C}_n + \sum_{i=1}^{M} \mathcal{C}_{known,i} + \mathcal{C}_{unknown} \tag{12}$$

Here, $\mathcal{C}_{vt}$ is the capacity used by the victim task vt; $\mathcal{C}_1, \ldots \mathcal{C}_n$ are the capacities used by n concurrent tasks; and $\mathcal{C}_{known,i}$, $i = 1 \cdots M$, and $\mathcal{C}_{unknown}$ denote capacities used by M known causes and any unknown cause. Using Equation 3, for victim task vt and concurrent tasks ct-s,

$$\mathcal{C}_{vt} = \frac{1}{\mathsf{RATP}_{vt}} \text{ and } \mathcal{C}_{ct} = \frac{1}{\mathsf{RATP}_{ct}} \text{ for } ct = 1 \cdots n \tag{13}$$

We abuse the notation to extend this concept also to other known and unknown causes as:

$$\mathcal{C}_{known,i} = \frac{1}{\mathsf{RATP}_{known,i}} \text{ for } i = 1 \cdots M \text{ and } \mathcal{C}_{unknown} = \frac{1}{\mathsf{RATP}_{unknown}} \tag{14}$$

Definition A.1. *The total* ***slowdown*** *of task vt in time interval δt due to unavailability of resource r is defined as:*

$$\mathcal{S}_{vt} = \frac{(\mathsf{RATP}_{vt} - \mathsf{RATP}^*)}{\mathsf{RATP}^*} \tag{15}$$

where, RATP_{vt} *is computed as per Equation (2).*

It is the deviation from ideal resource acquisition rate on host h and gives a measure of the excess delay incurred for unit resource in δt interval. The slowdown of vt will be zero when the entire resources is available only to the task vt.

The slowdown given in Definition 3.2 corresponds to the total blame to be attributed to (p_1) other tasks running concurrently with vt on h during its execution, and (p_2) other known or unknown factors. This gives another expression for slowdown:

$$\mathcal{S}_{vt} = \underbrace{(\sum_{ct=1}^{n} \beta_{ct \to vt})}_{p_1} + \underbrace{(\sum_{i=1}^{M} \beta_{known,i \to vt}) + \beta_{unknown \to vt}}_{p_2} \tag{16}$$

Here $\beta_{ct \to vt}$ is the blame assigned to each of the n tasks $ct = 1 \cdots n$ concurrently running with vt; $\beta_{known,i \to tt}$ gives the blame assigned to other known non-conflict-related causes that contribute to the wait time of the task vt. However these processes are identified and captured in iQCAR. Hence we categorize them as *known* processes. Finally, slowdown could be due to a variety of other causes which are either not known or cannot be attributed to any concurrent tasks like systemic issues (executors getting killed, external processes, etc), and so on; $\beta_{unknown \to tt}$ captures this value of slowdown due to such unknown factors.

B COMPUTATION OF BLAME BY RATPS

We now derive the blame values (β terms) in Equation (5) in terms of RATPs. First we discuss a simpler case to present the main ideas - when there is a full overlap of vt with all concurrent tasks. Then we discuss the general case with arbitrary overlap between ct-s and vt.

B.1 Full overlap of vt with concurrent tasks

Rewriting Equation (12) for $\mathcal{C} = \frac{1}{\mathsf{RATP}^*}$ from Equations (3), (13), and (14),

$$\frac{1}{\mathsf{RATP}^*} = \frac{1}{\mathsf{RATP}_{vt}} + \sum_{ct=1}^{n} \frac{1}{\mathsf{RATP}_{ct}} + \sum_{i=1}^{M} \frac{1}{\mathsf{RATP}_{known,i}} + \frac{1}{\mathsf{RATP}_{unknown}}$$

Multiplying by RATP_{vt} and subtracting 1 on both sides yield,

$$\frac{\mathsf{RATP}_{vt} - \mathsf{RATP}^*}{\mathsf{RATP}^*} = \sum_{ct=1}^{n} \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{ct}} + \sum_{i=1}^{M} \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{known,i}} + \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{unknown}}$$

The left hand side above represents the slowdown $\mathcal{S}_{vt}$ of vt given by Definition 3.2. Therefore,

$$\mathcal{S}_{vt}^{\delta t} = \underbrace{\sum_{ct=1}^{n} \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{ct}}}_{p_1} + \sum_{i=1}^{M} \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{known,i}} + \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{unknown}} \tag{17}$$

Each individual term inside the summation of p_1 is contributed by one of the tasks concurrent to task vt, and corresponds to blame attributable to a concurrent task ct in this interval. Comparing Equations (17) and (5) and assuming full overlap we get,

$$\beta_{ct \to vt}^{full\_overlap} = \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{ct}} \tag{18}$$

Similarly, for known and unknown factors,

$$\beta_{known,i \to vt}^{full\_overlap} = \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{known,i}} \text{ for } i = 1 \cdots M \tag{19}$$

$$\beta_{unknown \to vt}^{full\_overlap} = \frac{\mathsf{RATP}_{vt}}{\mathsf{RATP}_{unknown}} \tag{20}$$

B.2 Partial overlap of vt with concurrent tasks

The above derivation assumes an interval δt in which all concurrent tasks have a total overlap with vt. In practice, they overlap for different length of intervals as illustrated in Figure 5. So we divide the total duration $T = vt_{end} - vt_{start}$ of the execution time of task vt into small δt intervals such that in each δt time Equation (17) holds.

Let $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$ be the slowdown in each $m = \frac{T}{\delta t}$ interval of execution. The total slowdown of vt then is:

$$\mathcal{S}_{vt} = \sum_{k=1}^{m} \mathcal{S}_k$$

Substituting the value of slowdown $\mathcal{S}_k$ in the k-th interval using Equation (17), $\mathcal{S}_{vt}$ =

$$\sum_{k=1}^{m} \left[\sum_{ct \in \theta_k} \frac{\mathsf{RATP}_{vt}^{\delta t}}{\mathsf{RATP}_{ct}^{\delta t}} + \sum_{known \in \eta_k} \frac{\mathsf{RATP}_{vt}^{\delta t}}{\mathsf{RATP}_{known}^{\delta t}} + \frac{\mathsf{RATP}_{vt}^{\delta t}}{\mathsf{RATP}_{unknown}^{\delta t}} \right]$$

where, θ_k and η_k are the set of concurrent tasks and known factors respectively in the k-th interval impacting task vt. Note that the RATP values in the above equation depend on the intervals δt.

Rearranging the summations, we get the expression of blame for general overlaps as follows:

Proposition B.1. *The* ***blame*** $\beta_{ct \to vt}$ *for the contention caused for resource r by a concurrent task ct of a victim task vt on host h can be expressed as:*

$$\beta_{ct \to vt} = \left[\sum_{k=1}^{m} \frac{\mathsf{RATP}_{vt}^{\delta t}}{\mathsf{RATP}_{ct}^{\delta t}} \right] \tag{21}$$

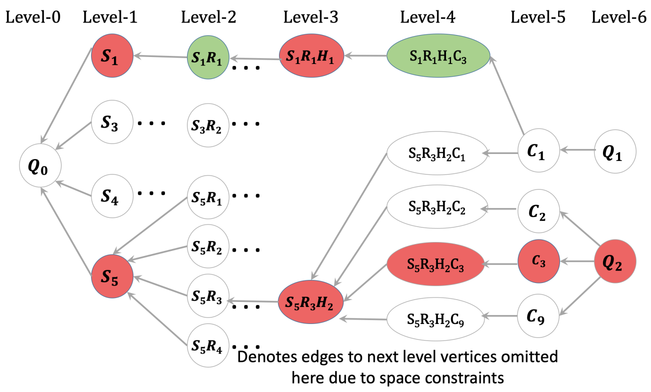


Figure 16: An instance of iQC-Graph **illustrating the impact on query depicted in Figure 1.**

C IQC-GRAPH CONSTRUCTION

iQC-Graph has 7 levels from Level 0 to Level 6. Each level in the graph represents an entity to which we assign the blame for the slowdown of a victim query. Level-0 represents victim queries; for every victim query vertex Q_i we add its victim stages as vertices in Level-1. Nodes in Levels 2 and 3 respectively capture the impacts coming from resources and hosts. That is, for each victim stage vertex $V_{i,j}$ in Level-1, we add five resource-level vertices (CPU, network, memory, IO, slots) at Level-2 to store the blame originating from them. For every Level-2 node, we add $\mathcal{H}$ vertices in Level-3 where each node represents the host on which the tasks of victim stage $V_{i,j}$ were executing and using resource r.

Nodes in Level 4 act as a bridge in connecting the concurrent stages (Level-5) with the host-level nodes at Level-3, and do not represent any culpable entity unlike other levels. Each Level-4 vertex captures the blame attributed to some concurrent stage $C_{m,n}$ of a concurrent query Q_n executing on host h and contending for resource r with victim stage $V_{i,j}$. Finally, we connect these *blame-attribution* vertices to the respective concurrent stages and their corresponding queries at Level 5 and Level 6 respectively. Figure 6 shows a subset of an example iQC-Graph for a single victim query Q_0 that captures the distribution of blame among two resources R_1, R_2, two hosts H_1, H_2, two concurrent stages C_1, C_2 of a single query Q_1, an *External-IO* process (a known factor), and an *Unknown* factor.

D IQC-GRAPH BY EXAMPLE

In the example of Q_3 shown in Figure 1, suppose the user selects Q_0 as the victim 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 stages, 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 network contention faced by stage s_5 was higher than the IO contention faced by stage s_1, and so on.

Suppose only the *trailing* tasks of stage s_1 executing on host Y faced IO contention due to data skew. Using iQC-Graph, a deep explanation tells the user that IO_BYTES_READ_RATP on host Y for these tasks of s_1 was much less compared to the average RATP for tasks of stage s_1 executing on other hosts. This insight tells the user that the slowdown on host Y for tasks of s_1 was not an issue due to IO contention. If the user lists the top-k nodes at Level-3 using our blame analysis API, she can see that the network RATP for tasks of stage s_5 on host X was the highest, and can further explore the Level-4 nodes to find the source of this disproportionality.

Since stage r_3 of culprit query Q_1, and stages u_5, u_7, u_9 of another concurrent query Q_2 were executing concurrently on host X with stage s_5 of Q_0, the user lists the top-k Level-4 vertices responsible for this network contention for s_5. The blame analysis API outputs stage u_7 of query Q_2 as the top source of contention. Figure 16 shows some relevant vertices for this example to help illustrate the levels in iQC-Graph.

E IQCAR USE-CASES

In this section, we present how iQCAR can be used to perform the following use-cases:

- **Finding Top-K Contentions for Victim Queries:** For any query, users can find answers for: (a) What is the impact of resource r for slowdown? This is computed by filtering the candidate explanations originating from all concurrent queries for the input resource r, and then aggregating the DOR values at Level-2. (b) What is the impact through host h for resource r? The candidate explanations with h and r on its paths are filtered and their DOR values at Level-3 are aggregated. Finally, (c) What is the impact through each culprit query or culprit stage for a host and resource combination? The candidate explanations for culprit query or culprit stage are filtered and the aggregate DOR at Level-5 or Level-6 respectively are output.
- **Identifying Slow Nodes and Hot Resources:** Performing top-k analysis on levels 2 (resources) and 3 (hosts) will yield the hot resource and its corresponding slow node with respect to a particular victim query. To get the overall impact of each resource or host on all victim queries, iQCAR provides an API to (i) *detect slow nodes*, i.e., for all explanation paths in the graph, groups all nodes in Level 3 by hosts and returns the total outgoing impact per host, and (ii) *detect hot resources*, i.e., - return the total outgoing impact per resource nodes in Level 2.
- **Detecting Culprit Queries:** To detect culprit queries, we find the top-k Level 6 nodes with highest total DOR towards all queries. As the framework incorporates impacts

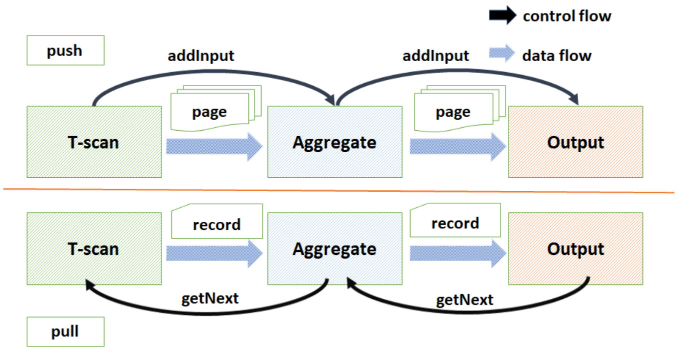


Figure 17: Pull vs Push based Query Engines

originating from other non-query sources, it enables an admin to rule out slowdown due to concurrency issues if the impact through these nodes is high.

F IMPLEMENTATION ON PRESTO

Presto is a distributed query engine whose execution model is similar to Spark. There are however some key differences that need to be considered before adapting iQCAR .

(1) Tasks vs Drivers: In Presto a task contains one or more parallel drivers which are the actual execution units. The invariant in Equation 12 is thus applicable to drivers in Presto (not Tasks). Adapting iQCAR required calculating and aggregating blame over drivers (tasks were bypassed and aggregation was done over all drivers of a stage on a host).
(2) Split Multiplexing on Threads: The TaskExecutor on worker nodes runs long running threads that process splits using round robin scheduling. A driver could run on different threads in different time quanta. This is different compared to other systems where a task runs entirely on one single thread.
(3) Pull vs Push: Presto Drivers use a push approach. An operator generates a page with getOutput and calls addInput on its dependent. This is different from the pull based approach, where getNext calls are chained in reverse direction of data flow. This is depicted in Figure 17

We adapted iQCAR to run on Facebook Presto 0.216 deployed on our cluster described in section 6. The same TPCDS dataset was stored in HDFS and accessed through Hive (version 2.33) using the Presto Hive Connector.

F.1 Supported Resources

In this section we describe our implementation to support metrics collection for all four system resources:

- CPU: Presto already captures the *totalCpuTime* and *elapsedTime* for every driver in its DriverStats. We further instrumented the code to additionally capture *cpulockwait* and *cpublockwait* time using the JMX API. The *OSSchedulingWait* is computed as described in section 5.2
- IO: Presto captures the *rawInputDataSize* for source operators. We instrumented the code to identify IO data based on the source operator types used in our workload (ScanFilterProject, PageSource and TableScan). As the execution follows a push based approach, IO blocked time does not equate to time between page results of source operator. In every interval we consider this wait time as blocking only if other operators in pipeline could not make progress due to missing input. This required instrumentation to the operator pipeline execution in the processInternal method of Driver.
- Network: Similar to IO, we use the *rawInputDataSize* metric of source operators but only when the operator is of type ExchangeOperator. The *networkBlockTime* is captured by instrumenting the ExchangeClient to identify response time of asynchronous HttpPageBufferClient requests. Due to the push based approach this time is considered as network blocked time only if other operators in the pipeline cannot make progress due to missing data in an interval.
- Memory: Presto manages application memory by splitting them into memory pools (General and Reserved). Queries consume memory from the pools with limits defined by configuration parameters. A query consumes three types of memory: user, revocable and system. The memory consumption of each driver is already captured in its *driverMemoryContext*. The time spent waiting for memory is captured by tracking the *memoryFuture* of operator contexts. This time is considered blocking only if all dependent operators in the pipeline cannot make progress.

F.2 Metrics Collection

Presto schedules drivers for small intervals (*splitRunQuanta*). While this provides an opportunity to all queued drivers to make progress, it also results in more dynamic resource interactions. To ensure that iQCAR captures these accurately, we captured the metrics with frequency close to *splitRunQuanta* but in a separate thread to ensure that driver runtime is not affected by metric collection.

F.3 Discussion

As discussed in section 5.5 iQCAR can work with a system like Presto by implementing the required metrics collection component. This specifically required three aspects (a) understanding its execution model (push vs pull, task vs driver etc) (b) instrumenting some missing metrics and (c) collecting the instrumented (and existing) metrics.