Dynamic Query Re-Planning using QOOP

Kshiteej Mahajan, *UW-Madison;* **Mosharaf Chowdhury,** *U. Michigan;*
Aditya Akella and Shuchi Chawla, *UW-Madison*

https://www.usenix.org/conference/osdi18/presentation/mahajan

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Dynamic Query Re-Planning Using QOOP

Kshiteej Mahajan[1] Mosharaf Chowdhury[2] Aditya Akella[1] Shuchi Chawla[1]
[1]*University of Wisconsin - Madison* [2]*University of Michigan*

Abstract

Modern data processing clusters are highly dynamic – both in terms of the number of concurrently running jobs and their resource usage. To improve job performance, recent works have focused on optimizing the cluster scheduler and the jobs' query planner with a focus on picking the right query execution plan (QEP) – represented as a directed acyclic graph – for a job in a resource-aware manner, and scheduling jobs in a QEP-aware manner. However, because *existing solutions use a fixed QEP throughout the entire execution*, the inability to adapt a QEP in reaction to resource changes often leads to large performance inefficiencies.

This paper argues for *dynamic query re-planning*, wherein we re-evaluate and re-plan a job's QEP during its execution. We show that designing for re-planning requires fundamental changes to the interfaces between key layers of data analytics stacks today, i.e., the query planner, the execution engine, and the cluster scheduler. Instead of pushing more complexity into the scheduler or the query planner, we argue for a redistribution of responsibilities between the three components to simplify their designs. Under this redesign, we analytically show that a greedy algorithm for re-planning and execution alongside a simple max-min fair scheduler can offer provably competitive behavior even under adversarial resource changes. We prototype our algorithms atop Apache Hive and Tez. Via extensive experiments, we show that our design can offer a median performance improvement of 1.47× compared to state-of-the-art alternatives.

1 Introduction

Batch analytics is widely used today to drive business intelligence and operations at organizations of various sizes. Such analytics is driven by systems such as Hive [5] and SparkSQL [19] that offer SQL-like interfaces running atop cluster computing frameworks such as Hadoop [4] and Spark [59]. Figure 1 shows the key layers of data analytics stacks today. At the core of these systems are

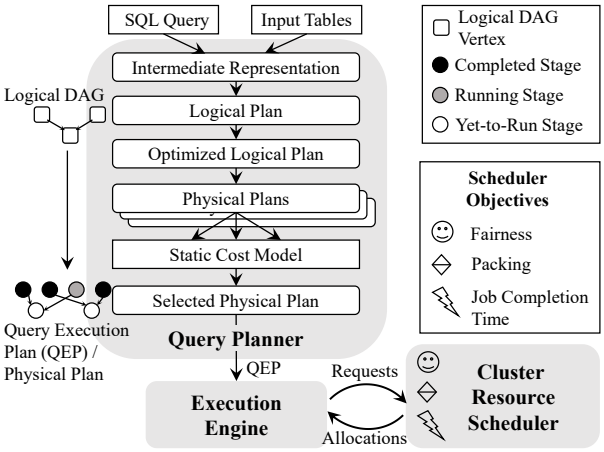


Figure 1: Traditional batch query execution pipeline.

query planners (QPs) , such as Calcite for Hive [3] and Catalyst for SparkSQL [19]. QPs leverage data statistics to evaluate several potential query execution plans (QEPs) for each query to determine an optimized QEP. The optimized QEP is a DAG of interconnected stages, where each stage has many tasks. An execution engine then handles the scheduling of these tasks on the underlying cluster by requesting resources from a scheduler. The scheduler allocates resources considering a variety of metrics such as packing, fairness, and job performance [35, 36, 61].

To improve query performance, existing works have primarily looked at optimizations limited to specific layers in the data analytics stack. Some of them [58, 61, 18, 34, 36, 35, 51] have focused on improved scheduling given the optimized QEP by incorporating rich information, such as task resource requirements, expected task run times, and dependencies. Others have considered improving the QP to take into account resource availability at query launch time (in addition to data statistics) to find good resource-aware QEPs [54, 55].

We argue that these state-of-the-art techniques fall short in *dynamic environments*, where resource availability can

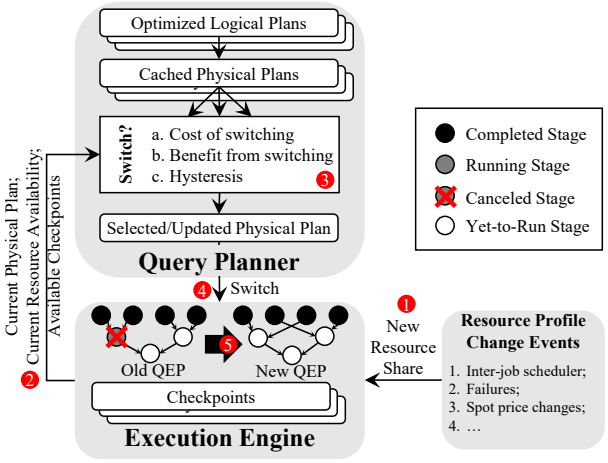


Figure 2: Dynamic replanning in action using QOOP. Omitted part of the query planner is similar to that of Figure 1.

vary significantly over the duration of a job's lifetime. This is because existing techniques are *early-binding* in nature – a QEP is pre-chosen at query launch time and the QEP's low-level details (e.g., the physical tasks, task resource needs, dependencies) are used to make scheduling decisions (which tasks to run and when). This fundamentally leaves limited options to adapt to resource dynamics. Our paper makes a case for *constant query replanning* in the face of dynamics. Here, a given job switches query plans during its execution to adapt to changing resource availability and ensure fast completion.

Dynamic resource variabilities can arise in at least two situations: (i) running multiple jobs on small private clusters, which is a very common use-case in practice [6]; and (ii) leveraging spot market instances for running analytics jobs, which is an attractive option due to the cost savings it can offer [45, 52, 62, 38]. We empirically study resource changes in these situations in Section 2.

To enable effective adaptation in these situations, we develop and analyze *strategies for query replanning*. We prove two basic results: (1) When dynamically switching QEPs, it is important for a query to potentially *backtrack* and forgo already completed work. Given imperfect knowledge of future resource availability, a query's performance can be arbitrarily bad without backtracking. (2) A *greedy algorithm* – which always picks a QEP offering the best completion time assuming current allocation persists into the future – performs well. We prove that the greedy algorithm has a competitive ratio of 4; the lower bound for any online algorithm is 2.

To realize the aforementioned replanning strategies in practice, we eschew the early binding in today's approaches. Instead, we propose a new system, QOOP, that has the following radically different division of labor and interfaces among the layers of analytics stacks (Figure 2):

- The cluster scheduler implements *simple* cluster-wide weighted resource shares and *explicitly* informs a job's execution engine of changes to its cluster share. The cluster share of a job is defined as the total amount of each resource divided by the number of active jobs. During a jobs's execution, our scheduler tracks a job's current resource usage – measured as the maximum of the fractions of any resource it is using – and allocates freed up resources to the job with the least current usage, emulating simple max-min fair sharing. Thus, the scheduler decouples the feedback about cluster contention – this helps queries replan and adapt – from task-level resource allocation, which is instantaneously max-min fair.

- When resource shares change *significantly*, the query planner compares a query's remaining time to completion based on its current progress against its expected completion time from replanning and switching to a different plan. It uses a model of task executions and available checkpoints in the execution engine to make this decision. It picks a better QEP to switch to (if one exists), and informs the execution engine of the new set of tasks to execute and existing ones to revoke.

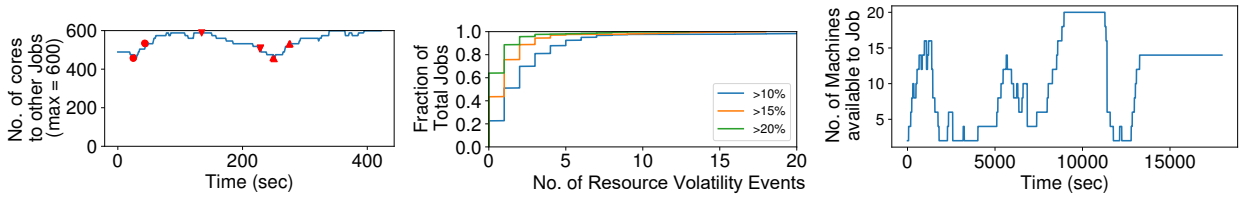
- The *execution engine* supports the query planner by informing it of the query's current progress and maintaining checkpoints of the query's execution from which alternate QEPs' computation can begin.

Overall, QOOP pushes complexity up the stack, out of cluster schedulers – where most of the scheduling complexity exists today – and into a tight replanning feedback loop between the query planner and the execution engine. We show that the resulting *late binding* enables better dynamic query adaptation.

We prototype QOOP by refactoring the interfaces between Hive, Tez, and YARN. Our evaluations on a 20-node cluster using TPC-DS queries show that QOOP's dynamic query replanning and simple scheduler outperform existing state-of-the-art static approaches. From a single job's perspective, QOOP *strictly* outperforms a resource-aware but static QP. For example, when resource profiles fluctuate rapidly, with high volatility, QOOP offers more than 50% of the jobs improvements of 1.47× or more; 10% of the jobs see more than 4× gains! We also use QOOP to manage the execution of multiple jobs on a small 20-node private cluster. We find that QOOP performs well on all three key metrics, i.e., job completion times, fairness, and efficiency, by approaching close to the individual best solutions for each metric.

2 Background and Motivation

In this section, we highlight multiple sources of resource dynamics in a cluster (§2.1), discuss the opportunities lost from not being able to switch a query's plan in response to resource dynamics (§2.2), and why the existing interfaces



(a) Jobs' view of cluster resources **(b)** Distribution of resource volatility **(c)** Number of machines reserved by a job in a spot market

Figure 3: Analysis of resource perturbations in a shared cluster and spot market. The gaps between each pair of the same symbols in (a) demarcate one resource volatility event.

between cluster schedulers, execution engines, and query planners make dynamic switching difficult (§2.3).

2.1 Resource Dynamics in Big Data Environments

Modern big data queries run in dynamic environments that range from dedicated resources in private clusters [25, 22] and public clouds [1] to best-effort resources put together from spot markets in the cloud [45, 52, 62, 38].

In case of the former, resources are arbitrated between queries by an inter-job scheduler [35, 30, 36, 61, 18]. As new jobs arrive and already-running jobs complete, resource shares of each job are dynamically adjusted by the scheduler based on criteria such as fairness, priority, and time-to-completion. Although in large clusters, such as those run by Google [25, 28] and Microsoft [22], individual job arrivals or departures have negligible impact on other jobs, most on-premise and cloud-hosted clusters comprise less than 100 machines [6, 9] and run only a handful of jobs concurrently. A 2016 Mesosphere survey [6] found that 96% of new users, and 75% of regular users use fewer than 100 nodes. A single job's arrival or completion in such scenarios can create large resource perturbations.

To better highlight resource perturbations in small clusters, we ran a representative workload on a 20-node cluster managed by Apache YARN. The cluster uses the Tetris [34] cluster scheduler, and it can concurrently run 600 containers at it's maximum capacity (1 core per container in a 600 core cluster). For our workload, we use the TPC-DS [12] workload, where jobs arrive following a Poisson process with an average inter-arrival time of 20 seconds. The average completion time per job is around 500s. We pick a job executed in the cluster and show it's view of cluster resources in Figure 3a. Specifically, we show the number of cores allocated (out of a maximum of 600) to all the *other* jobs running concurrently. During its lifetime, the job we picked experiences resource volatility – we call an x% increase or decrease in resource (number of cores in this case) over some period of time as an x% resource volatility. In Figure 3a, we identify 15% resource volatility within uniquely shaped red markers; e.g., the region between two solid red circles indicates one such 15% resource volatility. The job observes 3 such resource volatility events during its lifetime (identified within similarly shaped markers). To understand resource volatility as observed by different jobs for different resource volatility magnitudes (different values of x), in Figure 3b, we plot a CDF of the number of resource volatility events seen by each of the individual jobs in our workload for three values of x = 10%, 15% and 20%. We observe that almost 78% of the jobs experience at least one 10% resource volatility event during their lifetime, and 20% of the jobs see at least 4 resource volatility events of 10% or more.

At the other extreme, running jobs on spot instances – with their input on blob storage like Amazon S3 [2] – is becoming common because spot instances offer an attractive price point [45, 52, 62, 38]. However, cloud providers can arbitrarily revoke spot instances, which can cause perturbations in the number of machines available to a job. We now empirically examine the extent of such potential resource variations as experienced by a resource-intensive, batch job that runs for five hours. We use the spot-market price trend for `i3.2xlarge` instance type in Amazon EC2 cluster in the `us-west-2c` region for the time period from 17:00 UTC to 21:00 UTC for September 21, 2017. We also assume that the job has a budget of 5$/hour and that spot instances that were reserved at less than the current spot market price are taken away immediately. The spot instance prices typically update every minute. We use a simple cost-saving bidding strategy where the job progressively adds 2 spot instances every minute, provided the budget is not exceeded, by bidding at a price 5% over the current spot market price. Under such a bidding strategy and a budget of 5$, the maximum number of machines that the job gets is 20 and the minimum is 2. The number of spot instances available to the job over time is shown in Figure 3c. We make the following observations. First, the job experiences many perturbations in the number of machines, which is especially true with cost-saving bidding strategies. Second, the magnitude of perturbation is the largest around the 3

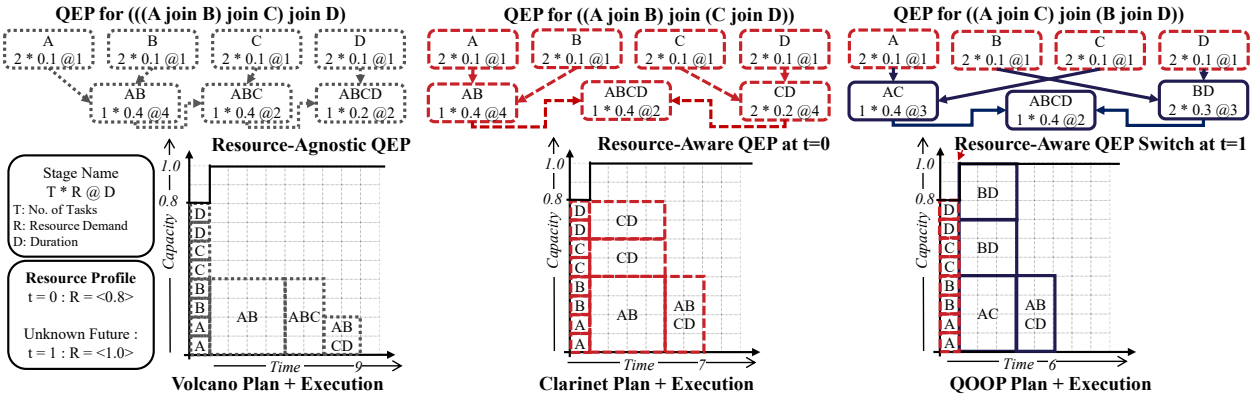


Figure 4: Comparison of query planners. The QEPs shown correspond to three different logical plans for the query - $A \bowtie B \bowtie C \bowtie D$. Volcano chooses the QEP on the left (as this plan has the least resource consumption) that completes at $t = 9$. Clarinet chooses an optimal resource-aware QEP (under static resources) at $t = 0$ that completes at $t = 7$; however, it ceases to be resource-aware when available resources change at $t = 1$. QOOP re-plans to switch to a new plan at $t = 1$ and completes the fastest at $t = 6$.

hour mark when the spot market instance price reaches a maxima of 0.5828$/hour and all but 2 machines are revoked. Finally, throughout the entire duration of the job, the job experiences 60, 53 and 40 resource volatility events of 10%, 15% and 20% respectively.

Other common sources of resource fluctuations include machine/rack failures, planned or unplanned upgrades, network partitions, etc. [21, 56].

2.2 Query Execution Today: Fixed Plans

Regardless of the extent of resource dynamics, existing approaches *keep the query plan fixed throughout the entire duration of a query's execution*. However, these approaches do vary in terms of what information they use during query planning and how they execute a query.

Resource-Agnostic Query Execution: A large number of today's data-analytics jobs are submitted as SQL queries via higher-level interfaces such as Hive [5] or Spark SQL [19] to cluster execution engines (Figure 1).

A cost-based optimizer (CBO) examines multiple equivalent logical plans for executing a query, and leverages heuristics to select a good plan, also called a query execution plan (QEP).[1] The QEP represents the selected logical plan and its relational operators as a *job* with a directed acyclic graph (DAG) of computation stages and corresponding tasks that will be executed by the underlying execution engine on a cluster of machines. Given the chosen QEP – also called the physical plan – the execution engine interacts with the cluster resource scheduler in a repeated sequence of resource requests and corresponding allocations until all the tasks in the physical plan of the job complete. *Crucially, the optimizer's heuristics are based on data statistics and not resource availability; thus, it is resource-agnostic.* An example is the Volcano query planner in Hive [5]. Figure 4 shows a Volcano-generated plan – a QEP corresponding to a "left deep" plan – that is preferred by the Volcano CBO based on data statistics.

Resource-Aware QEP Selection: Given the obvious inflexibility of resource-agnostic query optimization, some recent works [54, 55] have proposed resource-aware QEP selection. In this case, the CBO takes available resources into account before selecting a QEP and handing it over to the execution engine. While this is an improvement over the state-of-the-art, the execution engine still runs a fixed QEP even when resource availability changes over time. An example of a resource-aware planner is Clarinet [54]. As shown in Figure 4, the Clarinet plan is chosen based on the resources available at $t = 0$. When the resources change at $t = 1$, the static plan ceases to be the best.

Room for Improvement: Instead of sticking to the original resource-aware or -agnostic QEP throughout execution, one can find room for improvements by switching to a new QEP on the fly based on resource changes. For example, when the available resource increases at $t = 1$ in Figure 4, we can switch to a different join order – $(A \bowtie C) \bowtie (B \bowtie D)$ instead of $(A \bowtie B) \bowtie (C \bowtie D)$ – and further decrease query completion time.

Although this is a toy example, overall benefits of dynamic query re-planning improve with the complexity of query plans, magnitudes of resource volatility, and pathological fluctuations of resources due to unforeseen changes in the future (§6).

2.3 Scheduler Constraints on QEP Switching

Unfortunately, today's cluster schedulers and their interfaces with the execution engine and the query planner make resource-aware QEP switching challenging.

On the one hand, *existing schedulers provide little feedback to jobs about the level of resource contention in a cluster* – today, jobs simply ask the scheduler for re-

[1] Some optimizers consider a narrow set of resources, such as the buffer cache or memory, but ignore disk and network [5].

sources for runnable tasks and the scheduler grants a subset of those requests. Consequently, it is difficult for a job to know how to adapt in an informed manner to changing cluster contention or resource availability. One may think that jobs can infer contention by looking at the rate at which their resource requests are satisfied. However, such an inference mechanism can be biased by the resource requirements of the tasks in the currently chosen QEP instead of being correlated to the level of contention.

On the other hand, *scheduling decisions are tied to the intrinsic knowledge of job physical plan*. Schedulers are tasked with improving inter-job and cluster-wide metrics, such as fairness, makespan, and average completion time [30, 34, 27, 35, 36]. For example, DRF tracks dominant resources, which relies on the multi-dimensional resource requirements of physical tasks. Others [36, 34, 35] go further and combine resource requirements with the number of outstanding tasks and dependencies to estimate finish times using which scheduling decisions are made. The tight coupling of schedulers with pre-chosen QEPs constrains the scheduler to make decisions to match resources with the demands imposed by the pre-chosen QEP's tasks.

Overall, neither the job nor the scheduler has any way of knowing whether picking a different QEP with a very different structure and task-level resource requirements would have performed better – w.r.t. per-job or cluster-wide metrics – under resource dynamics.

3 QOOP Design

In this paper, we argue for breaking the constraints of fixed QEPs, and we make a case for continuous query re-planning by rethinking the division of labor between cluster schedulers, execution engines, and query planners. We first give an overview of our design (§3.1) and then present its three key components: a simple max-min fair scheduler (§3.2), an execution engine design to track additional states needed to speed up dynamic re-planning (§3.3), and a greedy QP that performs well with provable performance guarantees (§3.4).

3.1 Design Overview

The state-of-the-art approaches for improving query performance universally argue for pushing more complexity into the inter- and intra-job scheduling to achieve efficiency and improve job performance; by design, this prevents adaptation at the query level. Instead, to achieve replanning, we propose a significant refactoring. (1) We advocate having a simple max-min fair scheduler that effectively does "1-over-n" allocation of every resource across n jobs. (2) Jobs are informed as soon as their share changes due to changing n or machine/rack failures. (3) We push re-planning complexity *up the stack*, maintaining a *dynamic re-planning feedback loop* between the query planner and the execution engine: based on changes to the share, the planner – with help from the execution engine – determines if a better QEP exists and how to switch to it.

We choose this work division because each instance of an application framework today implements its own query planner and execution engine (e.g., both implemented in the Job Manager in case of frameworks using Apache YARN), whereas *all* jobs running in a cluster share the same centralized resource scheduler (i.e., the Resource Manager in Apache YARN). Our division of labor has the benefit of enabling many different applications with their intrinsic continuous re-planners to effectively run atop our simple cluster scheduler. For simplicity, our paper focuses just on re-planning batch SQL queries.

Figure 2 presents our architecture with the sequence of actions that take place on a resource change event: ❶ The cluster scheduler or the resource manager notifies the execution engine of its new resource share (§3.2). ❷ The execution engine, in turn, notifies the query planner of the current state, which includes the current QEP it is executing along with its progress, current resource availability it received from the scheduler, and the available set of checkpoints it is maintaining (§3.3). ❸ Given this information, the query planner must determine whether switching to a new plan is feasible (considering available checkpoints, cost of possible backtracking, and hysteresis) (§3.4). ❹ If the decision is yes, then it informs the execution engine of the new QEP. ❺ Finally, the execution engine will switch to the new QEP; if required, it will cancel some already-running stages and tasks.

Realizing dynamic re-planning raises a few key algorithmic questions. First, what is a good switching strategy when resources change? A simple and easy-to-implement choice is Greedy: i.e., always pick the QEP that offers the least estimated finish time assuming the new resource availability persists into the future. Does this offer good properties under arbitrary resource fluctuations? Second, switching from a QEP with partial progress to a new one that needs to be started from scratch necessarily wastes work. Is this "backtracking" necessary? In Section 4, we show that the simple Greedy approach performs well, and that backtracking is essential.

Before presenting the analysis, in Sections 3.3 and 3.4, we discuss key systems issues that arise in supporting greedy behavior with backtracking: How to estimate the relative runtimes of different QEPs? How to preserve work to support backtracking and leverage already computed work when switching to a new QEP? We start by outlining the functionality of our inter-job scheduler next.

3.2 Cluster Resource Scheduler

Our inter-job scheduler is simple (Pseudocode 1). For each job, our scheduler tracks the job's current (weighted) share in every resource dimension, i.e., the total fraction

of the resource that all currently running tasks of the job are using. The scheduler computes the *current share* of the job as the maximum of these fractions taken over all resources. When a resource is freed on a machine, our scheduler simply assigns it to the job with the lowest current share that can run on that machine, emulating simple instantaneous max-min fair allocation of resources across jobs, similar to [14, 30]. This is shown in lines 4–8. We are algorithmically similar to DRF, but differ in API. When resources become available DRF allocates to the job with least dominant-share; QOOP informs each job of its dominant share on resource-change events.

To enable re-planning, we introduce two changes to the interface between the scheduler and the execution engine. First, we do not require the execution engine to propagate the entire QEP to the cluster scheduler. Decoupling the QEP from the resources assigned to a job has the desirable property that the execution engine can change the QEP without affecting its fair share of resources, which is not the case for the state-of-the-art techniques [30, 36, 35].

Second, we introduce feedback from the cluster scheduler to the execution engine (line 9 in Pseudocode 1). Whenever the current cluster share of a job changes, the scheduler informs the job's execution engine. The cluster-wide fair-share informs each job of its minimum resource share given the current contention in the cluster. This acts as a *minimum resource guarantee* for the query planner when determining whether to re-plan in order to finish faster. In fact, any scheduler that can offer feedback in the form of an eventual minimum resource guarantee of resources to each job is compatible with QOOP.

3.3 Execution Engine

We discuss how job execution engine redesign can enable query re-planning, specifically backtracking.

Task Execution: Given a job QEP DAG (i.e., the output of a query planner), the execution engine executes tasks by interacting with the cluster scheduler while maintaining their dependencies. To determine the order of task execution, it can simply traverse the DAG in a breadth-first manner [59, 19] or use a multi-resource packing algorithm such as Tetris [34]. In QOOP, we use Tetris.

Checkpointing for Potential Switching Points: On any multi-resource update from the cluster scheduler, the execution engine relays the updated resource vector to the query planner to evaluate the possibility of switching to a different QEP. Determining whether to actually switch to a new QEP relies on multiple factors (§3.4). A major one is finding the suitable point(s) in the currently executing DAG to switch from. One may consider that switching from the currently executing stage or its immediate parent stage(s) would suffice. However, we prove in Section 4 that *backtracking* to ancestor stage(s) is essential for competitively coping with unknown future resource changes.

Pseudocode 1 Cluster Scheduler

```
1:  𝕁                          ▷ active jobs prioritized by lowest current share
2:  R⃗                          ▷ total cluster resource capacity
3:  U⃗                          ▷ consumed cluster resource portion

4:  procedure MaxMinFairScheduler
5:      pick first J ∈ 𝕁        ▷ triggered when R⃗ − U⃗ > 0⃗
6:      allocate demand D⃗_i ∈ J s.t. max_{i,m} D⃗_i · (R⃗_m − U⃗_m)
7:      update 𝕁
8:  end procedure

9:  procedure ResourceFeedback(Event 𝔼)
10:     𝕁 = 𝕁 ⊕ getJobChanges(𝔼)
11:     R⃗ = R⃗ ⊕ getResourceChanges(𝔼)
12:     fairShare = R⃗ / |𝕁|
13:     for all J_k ∈ 𝕁 do
14:         SendResourceFeedbackUpdate(J_k, fairShare)
15:     end for
16: end procedure
```

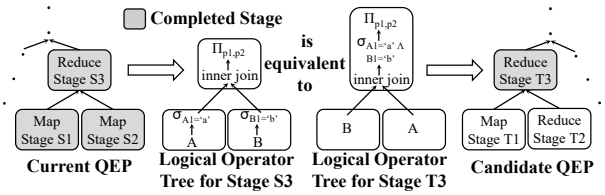


Figure 5: Progress propagation. First, we obtain logical operator trees for stages $S3$ and $T3$ from provenance. Stages $S3$, $T3$ are deemed equivalent as their logical operator trees are equivalent.

Consequently, QEP switching may not just re-plan the future stages of the query, but it requires the ability to checkpoint past progress and switch to a different QEP from an ancestor stage that was executed in the past. To enable this, the execution engine needs to checkpoint past progress for all the different QEPs it has executed thus far. Each checkpoint includes the intermediate outputs of completed tasks. Note that checkpointing of intermediate data is common in modern execution engines – disk-based frameworks write intermediate data to disks [25, 7], whereas in-memory frameworks periodically checkpoint to avoid long recomputation chains [59, 60]. QOOP can use this existing checkpointing.

Switching the QEP: The call back to the query planner (upon resource updates) is asynchronous. While the query planner is evaluating possible alternatives, the execution engine continues on with the current plan. When the query planner suggests a change, the execution engine revokes the resource requests for runnable tasks not belonging to the new QEP. Additionally, the execution engine may

abort running tasks not belonging to the new QEP. Thereafter, the execution engine resumes running tasks from the most-recent set of checkpoints for the new QEP.

3.4 Query Planner (QP)

In Section 4, we show that effective re-planning requires backtracking and that a greedy approach to re-planning results in a competitive online algorithm. Here, we present the details of how our query re-planner implements greedy re-planning by leveraging backtracking.

We introduce two key changes to the design of traditional QPs; neither requires extensive modifications. First, instead of discarding intermediate computations to explore and choose a particular QEP, we generate and cache several candidate QEPs. The cached QEPs later aid us in dynamic query re-planning. We also annotate each QEP with provenance, which consists of the original logical plan the QEP was derived from and the list of logical operators associated with each stage of the QEP. Figure 5 shows the provenance of each stage of a QEP.

Second, unlike traditional QPs [3, 54], our QP is made aware of the underlying resource contention to accurately predict runtimes for each QEP and greedily switch to the QEP with minimum completion time. To do so, we extend the interface between the QP and the execution engine so that the QP receives parameters to its dynamic cost model – the current resources available to the job (the share that the execution engine obtains from the cluster scheduler), the intra-job scheduling logic (packing), the progress of the current QEP and the available set of checkpoints.

Whenever the query planner receives a notification about resource changes from the execution engine, it triggers a cost-based optimization that involves predicting the completion times of all the QEPs and greedily switching to the QEP with earliest completion time. There are two steps to evaluate a particular QEP: progress propagation and completion time estimation.

Progress Propagation: To evaluate a candidate QEP, the QP first evaluates the work in the candidate QEP that is already done by the currently running QEP. It does so by identifying common work between the tasks of the candidate QEP, the running tasks of the current QEP, and the current set of checkpoints. We refer to this as *progress propagation*, and it is crucial in evaluating which candidate QEP to switch to and where to execute it from.

To identify common work as part of progress propagation, we identify equivalence between the stages of a candidate QEP and the set of checkpointed stages and the current running stages of the current QEP. To evaluate equivalence between two stages we generate the stages' logical operator trees using the provenance associated with each QEP. Two stages are deemed equivalent if their logical operator trees are equivalent. Equivalence of logical operator trees is evaluated using standard relational algebra equivalence rules. This is illustrated in Figure 5.

Completion Time Estimation: Next, we perform a simulated execution of the remaining tasks in the candidate QEP being evaluated (i.e., candidate QEP tasks whose work is not captured in the currently running QEP). Using the scheduling algorithm of the intra-job scheduler, i.e., Tetris, the remaining tasks are tightly packed in space and time given the current available resources. This yields an estimate for this QEPs completion time assuming that the current resource availability will persist in the future.

After evaluating the completion times of all candidate QEPs, query planner triggers a query plan switch if it finds a QEP that finishes faster than the currently running QEP. To avoid unnecessary query plan flapping, we add *hysteresis* by having a threshold on the percentage improvement of the query completion time – a query plan switch is triggered only if improvements exceed this threshold.

In case of a switch, the query planner sends the new QEP to the execution engine. This QEP is modified from its original form so that the DAG now contains the checkpoints as input stages, marks the running stages it shares with the running stages of the current QEP, and identifies the dataflow from these to the remaining stages.

4 Analysis

We now present analysis of the query planner (QP; Section 3.4). Each query has several alternative query execution plans (QEPs). We motivate the choices made in the query replanning algorithm regarding *why, when and which* QEP to switch to during the execution of a query in response to the resource allocations made by the scheduler to the query. This is an online algorithm since it operates without the knowledge of future resource allocations. We analyze the performance of our online algorithm in the form of its competitive ratio. Our goal is to argue that our online algorithm performs well no matter the sequence of resource allocations made to the query. We will compare our online algorithm's performance against an hindsight optimal (a.k.a. offline) algorithm which chooses the single best QEP knowing the entire sequence of resource allocations made to the query. The competitive ratio is the ratio of the performance of the online algorithm to that of the hindsight optimal (a.k.a. offline) algorithm. We provide a precise measure of comparison shortly (Section 4.3).

4.1 Notation and Assumptions

Notation: We represent each QEP as $a \times b$. This denotes a QEP with a bag of b tasks, each task needing a resource-units (e.g., number of cores) and each task completing in 1 step. The total work for this QEP is denoted by w and is equal to ab.

Assumptions: For the upper and lower bounds on performance, we assume an adversarial scheduler that can look at the algorithm's choices in the previous steps and

change future resource allocation in a worst-case manner. We require that the QP has the ability to *backtrack* a QEPs execution i.e., the QP can checkpoint each completed task in a QEP and any completed task need not be re-executed when the QP decides to switch back to and resume the execution of that QEP. We also assume that backtracking does not incur any overheads; in other words that our analysis ignores system-level costs (time spent and compute/memory used) in writing checkpoints and reading from checkpoints during a QEP switch.

4.2 Motivating Example

We motivate *why* QEP switching and specifically backtracking is necessary to obtain a bound on the performance of our online algorithm.

Our toy examples, with large work-differences in QEPs, serve to show that if the online algorithm does not make good decisions then its performance can become unboundedly worse.

Example 4.1.
QEP switching is necessary. *Consider a query with two QEP choices: the first one being* 2×2 *and the second one being* 1×100*. Suppose that the scheduler starts by giving the query* 2 *resource-units in the first step. We also suppose that the query cannot switch QEPs.*
CASE-1: *If the query starts running the* 1×100 *QEP, the scheduler gives it another* 2 *resource-units in the second step. With this allocation, the optimal choice would be to run the* 2×2 *QEP, finishing in two steps and performing only* 4 *units of work. The online algorithm instead performs* 100 *units of work if it continues to use the* 1×100 *QEP.*
CASE-2*: On the other hand, if the query starts running the* 2×2 *QEP, the scheduler switches to a resource allocation of* 1 *resource-unit second step and onwards. Now the* 2×2 *QEP is stalled. Unless the algorithm switches to the* 1×100 *QEP, it is unable to finish.*
QEP backtracking is necessary. *Backtracking helps* avoid stalling, ensures fast completion, and bounds wasted work. *We continue the previous example. As before, the scheduler continues to be adversarial. It allocates* 1 *and* 2 *resource-units in the next step whenever the query is executing* 2×2 *and* 1×100 *QEP in the current step, respectively. Also, we now suppose that the query has the ability to switch QEPs but not backtrack i.e., no ability to checkpoint and resume QEPs from checkpoint.*
We continue from where we left-off in the previous example i.e., CASE-2 *where the query is executing the* 2×2 *QEP and the scheduler allocates* 1 *resource-unit in the second step. With the ability to switch, to avoid stalling, the query switches to the* 1×100 *QEP in the second step. Without backtracking, the query has to restart execution of* 1×100 *QEP from the beginning. Now on switching to the* 1×100 *QEP, the adversarial scheduler gives the query* 2 *resource-units third step onwards. This leads us back to* CASE-1*. If the QEP continues with the* 1×100 *QEP it leads to* slower completion.
If instead the query switches back to 2×2 *QEP in the third step, without backtracking the QEP restarts execution from the beginning and the adversarial scheduler gives the query* 1 *resource-unit fourth step and onwards. This is* CASE-2 *all over again. We can now see that, without backtracking, the query flips between* CASE-1 *and* CASE-2 *and* stalls infinitely with unbounded wasted work.
Even if the query decides to limit wasted work by stopping the switch to 2×2 *QEP, complete execution of* 1×100 *QEP to completion leads to* 100 *units of additional work and* 100 *additional steps. This leads to* slower completion *as in* CASE-1.
If the query could backtrack – we would have only one additional task to run from the 2×2 *QEP in the third step and the query would complete execution in the third step with just* 1 *units of wasted work.*

4.3 Competitive Ratio

A natural way to compare the performance of our algorithm against the hindsight optimal algorithm is to compare the time each algorithm takes to complete the query. As the next example shows, this is not a meaningful comparison, because the scheduler has the power to starve the online algorithm after a single bad choice.

Example 4.2. Starvation. *Consider the above example again. As before, the scheduler starts by giving the query* 2 *resource-units in the first step. If the query starts running the* 1×100 *QEP, the scheduler gives it another* 2 *resource-units in the second step, and then gives no more resources to this query in subsequent steps. Regardless of whether the query continues running the* 1×100 *QEP or switches to the* 2×2 *QEP in the second step, the query is unable to finish the work and stalls. Its completion time is unbounded. With the same allocation of resources, the hindsight optimal algorithm could have finished the query by just running the* 2×2 *QEP.*

On the other hand, say the query starts running the 2×2 *QEP and the scheduler gives* 1 *unit resource for the next* 99 *steps and then gives no more resources. Once again no matter what the online algorithm does, it cannot complete the query. However, the hindsight optimal algorithm would have been able to complete the query given these resources.*

In each of the cases in the above example, the scheduler could stall the query for an unlimited time, whereas the hindsight optimal algorithm terminates in bounded time. In order to allow for some wasted work due to the online nature of the algorithm, the scheduler must provide more resources to the online algorithm than just the minimum necessary for the hindsight optimal algorithm.

Pseudocode 2 Online Query Planning Algorithm

Input n QEPs, $a_i \times b_i$, with $a_1 < a_2 < a_3 < \cdots < a_n$

1: Let $w_i = a_i b_i$ denote the total work of QEP i.
2: **for all** $i \in [n]$ **do**
3: **if** $w_i > \frac{1}{2} w_{i-1}$ **then** remove QEP i from the list.
4: **end if**
5: **end for**
6: At every step, given the current resource allocation a, consider all QEPs with $a_i \leq a$. Of these, run the QEP with the least remaining processing time, breaking ties in favor of the QEP with the smallest a_i.

To formalize this, we will compare the completion time of the online algorithm to that of an hindsight optimal algorithm that is required to perform extra work.

Definition 4.1. Competitive Ratio. *We say that an online QEP selection algorithm achieves a competitive ratio of α if for any query and any sequence of resource allocations, the completion time achieved by the online algorithm is at most equal to the completion time of an offline optimum that runs α back-to-back copies of the query.*

We note that α above does not have to be an integer.

4.4 Bounds for the Competitive Ratio

We show that no online algorithm can achieve a competitive ratio < 2. Proofs for the theorems below can be found in extended version of QOOP [47].

Theorem 4.1. *No online query planning algorithm can achieve a competitive ratio of $2 - \varepsilon$ for any constant $\varepsilon > 0$ when the resource allocation is adversarial.*

Our query planning algorithm corresponding to the simplifying assumptions in Section 4.1 is formally described above. It is greedy and at every step runs the QEP with the least remaining completion time with the assumption that the resource allocation persists forever. Also, it is "lazy" as it switches QEPs only when the resource allocation changes. Our overall approach in Sections 3.4 and 3.3 is a generalization of this algorithm for complex queries.

We prove that this algorithm is competitive:

Theorem 4.2. *The online greedy query planning algorithm described above achieves a competitive ratio of 4. Further, if the QEPs satisfy the property that every pair of QEPs is sufficiently different in terms of total work, in particular, $w_i \leq \frac{1}{2} w_{i-1}$ for all $i > 1$, then the competitive ratio is ≤ 2, matching the lower bound.*

We note that constant competitive ratio implies that the performance of our online query planning algorithm is independent of the nature of workloads or the environment.

5 Implementation

Implementation of QOOP involved changes to Calcite [3], Hive [5], Tez [7], and YARN [53]. QOOP's implementation took ~13k SLOC. The majority of our changes were in Tez mostly devoted to dynamic CBO module we elaborate upon shortly.

Hive and Calcite: Hive uses the Volcano query planner implemented in Calcite to get a cost-based optimized (CBO) plan. We add the ability to cache several logical plans in Calcite during its plan evaluation process and make changes to Hive to fetch multiple physical plans (i.e., Tez QEPs). Also, we make changes to annotate each QEP with provenance—the set of logical relational operators associated with each stage of the QEP. We widened the RPC interface from Hive to Tez, to push multiple QEPs to Tez as part of a single job.

Tez and Yarn: To enable dynamic query plan switching we added modules to Tez that are responsible for (i) accounting checkpoints to enable backtracking; (ii) dynamic cost-based optimization to make Tez QEP switching decisions; (iii) runtime QEP changes to realize QEP switching; and (iv) the RPC mechanism from YARN to Tez to give resource feedback (i.e., resource updates about the dynamic "1-over-n" share of resources).

Any resource change event from YARN triggers our dynamic CBO module that evaluates all QEPs. This module first propagates progress using provenance and estimates completion time of each QEP via simulated packing in the available resource share (§3.4). Our CBO relies on estimates of tasks' resource demands—CPU, memory, disk, and the network—and their durations. Peak resource estimates are based on prior runs for each QEP. We use these peak resource estimates to decide the container request sizes for tasks in the currently executing QEP.)

Checkpointing for backtracking and runtime changes to the QEP involve changes to the QEP, Vertex, and Task state machines in Tez. All checkpointing state is maintained at the Tez QEPAppMaster—which keeps the file handle of task output after every task completion event. For QEP switching, we added the `SWITCHING` state to the QEP state machine. On a resource change event a QEP is forced from `RUNNING` to `SWITCHING`. Any running tasks of the QEP continue running in this state but the launch of any new vertex (and hence its tasks) is prevented in this state. The QEP switches to `RUNNING` state after, if at all, QEP switching happens. During a QEP switch, the set of runnable Vertices is re-initialized to those from the new QEP. The Vertex definition is changed so that the inputs for the tasks spawned by any runnable Vertex points to the appropriate checkpoint.

6 Evaluation

In this section, we evaluate QOOP in situations with varying degrees of resource variabilities. We examine both

the performance of an individual query using QOOP's replanning as well as overall performance when multiple queries run atop QOOP.

We start by studying the execution of a single job, subjecting it to real resource change events or *resource profiles*. Specifically, in these *micro-benchmarks*, our focus is on answering the following key question: *does QOOP's dynamic query re-planning improve a job's completion time when compared to static, early-binding approaches?*

Next, we evaluate the key system components of QOOP – backtracking, overheads of QEP switching, robustness to errors in the task estimates, and hysteresis.

Finally, we consider a small private cluster, where QOOP is used to manage the execution of multiple jobs. We evaluate QOOP by running multiple jobs on the testbed, wherein job arrivals and completions can lead to large resource perturbations. This *macro-benchmark* addresses the question: *does QOOP's simple cluster scheduler and dynamic query re-planner approach improve system-wide objectives when compared against systems with complex schedulers and static query planners?*

6.1 Experimental Setup

Workloads: Our workloads consist of queries from the publicly available TPC-DS [12] benchmark. We experiment with a total of 50 queries running at a scale of 500, i.e., running on a 500GB dataset. [2] For micro-benchmarks, we focus on the perspectives of individual queries. For macro-benchmarks, each workload consists of jobs drawn at random from our 50 queries and arriving in a Poisson process with an average inter-arrival time of 80s. [3]

Cluster: Our testbed has 20 bare-metal servers – each machine has 32 cores, 128 GB of memory, 480 GB SSD, 1 Gbps NIC and runs Ubuntu 14.04. For micro-benchmarks, we evaluate QOOP under different realistic resource profiles, as elaborated later in this section. In such experiments, we provide as much resources from the cluster to each job over time as dictated by the resource profile. Specifically, whenever there is an increase in the amount of resources in the resource profile we make available to the job corresponding number of containers, whereas whenever there is a decrease in the amount of resources in the resource profile we immediately revoke equivalent number of containers and fail any tasks running on them.

For macro-benchmarks, we run our entire collection of jobs across the entire cluster. At its maximum capacity, the cluster can run 600 tasks (containers) in parallel.

Baselines: In micro-benchmarks, we compare QOOP's query planner against static query plans obtained from the *Clarinet QP*, which is a resource-aware QP implemented in Hive [54] that improves upon Volcano. We only compare against *Clarinet QP* as it outperforms Volcano. We adapted Clarinet to our setting to choose a QEP that minimizes completion time using resource estimates just before query execution begins. It represents the performance upper-bound of fixed-QEP approaches.

In macro-benchmarks, we compare QOOP – dynamic query planner on top of our simple max-min fair scheduler – against the following approaches on the three system-wide objectives of fairness, job completion time, and efficiency: (1) *DRF*: The default DRF multi-resource fair scheduler [30] in conjunction with Hive's default Volcano QP; (2) *Tetris*: A multi-resource packing scheduler [34] with Volcano; (3) *SJF*: Shortest-Job-First scheduler [27] with Volcano; (4) *Carbyne*: A meta-scheduler that leverages DRF, Tetris, and SJF [35] with Volcano; (5) *DRF+Clarinet*: DRF with the Clarinet QP [54]; (6) *Carbyne+Clarinet*: Carbyne scheduler with Clarinet QP.

These reflect combinations of query planners that differ in whether they are resource-aware with schedulers that differ in the complexity of information they leverage in making scheduling decisions.

Metrics: Our primary metric to quantify performance improvement using QOOP is improvement in the average *job completion time (JCT)*: $\frac{\text{(Average) JCT of an Approach}}{\text{(Average) JCT of QOOP}}$

Additionally, in multi-job scenarios, we consider *Jain's fairness index* [42] to measure fairness between jobs, and *makespan* (i.e., when the last job completes in a workload) to measure overall resource efficiency of the cluster.

6.2 QOOP in Micro-Benchmarks

QOOP has two core components: the dynamic query replanning logic for a single query (Sections 3.3 and 3.4), and the simple cross-job cluster wide scheduler (Section 3.2). We study the two separately, with this section focusing on the former using micro-benchmarks.

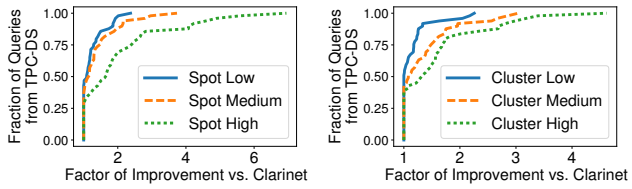
Specifically, in these micro-benchmarks, we ask: given a certain resource change profile, how well does a single query perform from using QOOP's query replanning algorithms? We study QOOP under two classes of resource change profiles, *spot instances* and *cluster resources* .

6.2.1 Spot Markets Resource Profiles

We obtained a 5-hour spot market price trend for `i3.2xlarge` instance type in Amazon EC2 cluster in the `us-west-2c` region for the time period from 17:00 UTC to 21:00 UTC for September 21, 2017. We infer the resource profile for the spot market price trend by applying the bidding strategy described in Section 2. We then divide the entire resource profile into "low", "medium", and "high" regions by time. To do so, we divide the entire resource profile into 10 minute regions and calculate the maximum increase or decrease in the resources in this 10

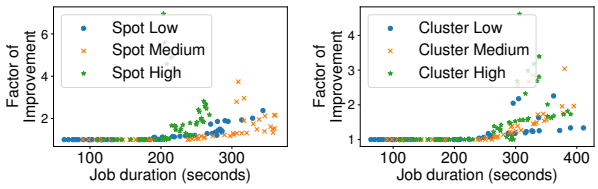
[2] We cached plans obtained while exploring QEPs in the Volcano planner, and retained plans with significant differences in cost according to Volcano's cost model. We used the first 50 TPC-DS queries that gave the most number of QEP alternatives.

[3] Google cluster trace [8] analysis on 20-machine sets yielded an average job inter-arrival time of 80s.



(a) Spot market **(b)** Shared cluster

Figure 6: Improvements using QOOP w.r.t. Clarinet under resource variations observed in different resource profiles.



(a) Spot market **(b)** Shared cluster

Figure 7: Improvements vs. job durations using QOOP w.r.t. Clarinet under different resource profiles.

minute region. We call an x% increase or decrease in resource in atleast one of the resource dimensions (compute or memory) over some period of time as an x% resource volatility. If the maximum resource volatility in resources is less than 10% then we classify this region as "low", if it is between 10% and 20% then we classify this region as "medium", if it is greater than 20% then it is classified as "high". We also refer to these as having "low", "medium" and "high" resource volatility . We then run each of our 50 TPC-DS queries individually against each of these three resource profiles – "low", "medium", and "high" – using both QOOP and Clarinet. For each query run with a particular resource profile type, we pick 10 different randomly selected regions of that particular profile type and report the mean from these 10 runs.

We plot the CDF of QOOP's improvements over Clarinet for the three resource profiles in Figure 6a. We see that QOOP *strictly* outperforms Clarinet, with its gains improving with increasing resource volatility – overall, 58%, 62% and 66% of the jobs experience faster completion times in each of "low", "medium", and "high" profiles, respectively. Median improvements for the "low", "medium" and "high" profiles are respectively 1.08×, 1.11× and 1.47×. For "high" profiles, 10% of jobs see gains $> 4\times$! We also note that 34% of the jobs show no improvements over Clarinet even with "high". On further analysis, we found that these jobs are queries in the TPC-DS workload that are either (i) less complex queries with lesser number of joins, or (ii) queries with short durations. Less complex queries may lack attractive alternative QEPs, whereas short queries may miss out on resource perturbations. This limits opportunities for re-planning and improvement. We dig deeper into these issues later in this section.

6.2.2 Shared Cluster Resource Profile

Similar to the spot market scenario, we generate three different resource profiles for the shared cluster scenario described in Section 2. Following a similar methodology, we identify "low", "medium" and "high" resource volatility periods, and we run each of the 50 queries.

As before, we plot the CDF of QOOP's improvements in Figure 6b. We see the trends similar to that of the spot market trace – overall, 56%, 58% and 60% of the jobs complete faster in "low", "medium", and "high" profiles, respectively. The median improvements in the three profiles are 1.08×, 1.11× and 1.20×, with higher performance improvements in greater resource volatility scenarios; for the "high" profile, 10% of jobs see gains $> 3.3\times$.

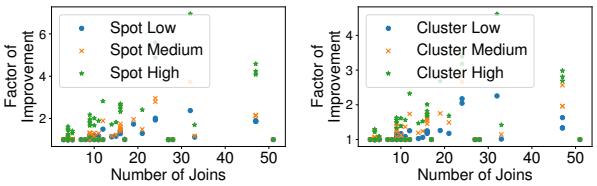
In both the spot instance and cluster profiles, gains are higher for profiles with higher volatility. In other words, QOOP's dynamic replanning is most effective relative to static query plans when resource volatility is at its highest. Also, the improvements for spot market and shared cluster, while similar for "low" and "medium", differ on the "high" resource profiles. We attribute this to spot market "high" resource profiles experiencing 7% larger magnitudes of resource changes at median than that of the shared cluster.

6.2.3 Delving into Improvements

Next, we take a deep dive into the aforementioned scenarios to understand when QOOP offers the greatest/least improvements. We study the impact of job duration, complexity, and the number of QEP switches that occur.

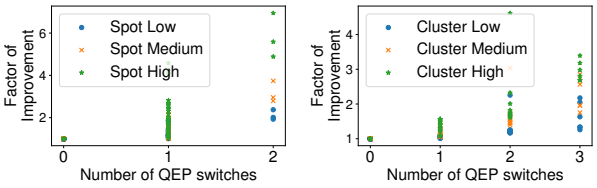
Job Durations vs. Observed Gains: The improvements in per-job performance due to QOOP as a function of job duration is shown in Figures 7a and 7b for the spot market and cluster resource profiles, respectively. Both figures also show results for the "low", "medium" and "high" volatility profiles using different-colored dots. In both cases, QOOP's benefits increase with increasing job durations. This is because longer jobs receive more opportunities for switching query plans and the comparative overhead of a switch of a longer job is smaller w.r.t. its completion time. Nevertheless, some shorter jobs benefit from QOOP in case of higher resource volatility.

Job Complexity vs. Improvement: Figures 8a and 8b show improvements obtained with QOOP as we increase query complexity for the spot market and cluster profiles, respectively. We measure query complexity in terms of the number of join operations in the query. We make two observations. First, increased query complexity generally correlates with increased gains. This is because the number of alternate query execution plans is higher with a greater number of joins. Second, keeping complexity



(a) Spot market **(b)** Shared cluster

Figure 8: Improvements vs. query complexity (number of joins) using QOOP w.r.t. Clarinet under various resource profiles.



(a) Spot market **(b)** Shared cluster

Figure 9: Improvements vs. number of QEP switches using QOOP w.r.t. Clarinet under various resource profiles.

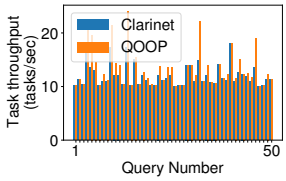


Figure 10: Improvements w.r.t. Clarinet vs. number of QEP switches in spot market.

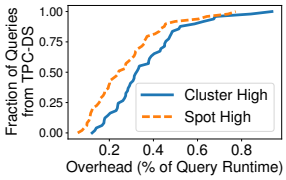


Figure 11: Overheads due to QEP switches measured as % of a job's completion time.

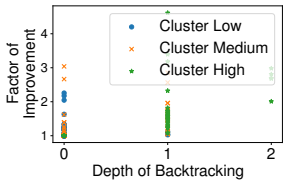


Figure 12: Improvements vs. depth of backtracking.

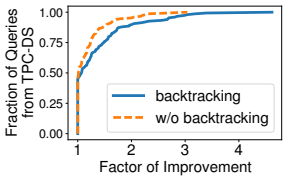


Figure 13: Improvements with and without backtracking.

constant, higher volatility results in the highest factor of improvement (as indicated above).

QEP switches vs. Improvement: Figures 9a and 9b shows the trend between improvements and number of runtime QEP switches. First, we see that an increase in the number of query execution plan switches correlates with increased gains. Second, keeping the number of switches constant, higher volatility results in the highest factor of improvement. In general, the greater flexibility a query intrinsically has in terms of multiple alternate plans together with the flexibility QOOP offers in switching to these plans results in a higher degree of improvement.

Task Throughput: Finally, we consider how fast QOOP helps the query complete tasks over time. We measure task throughput as the average number of tasks of the job executed per second; higher implies better utilization. In Figure 10 we show the task throughput of QOOP and Clarinet across queries. The number of tasks per second in the case of QOOP exceeds Clarinet by $\sim 24\%$ in the average case. Further analysis showed that an increase in the number of resources available leads QOOP to switch to query execution plans that favor more parallelism (i.e., "bushy" joins) and contributes to increased utilization.

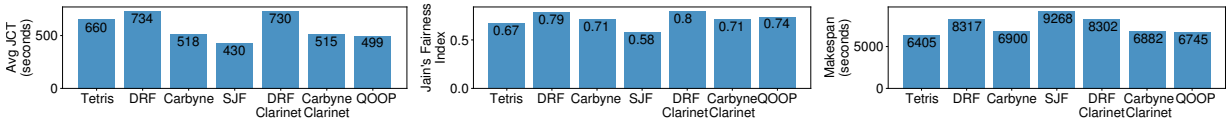
6.3 Impact of Various QOOP Features

In this section, we study the effect of different aspects of QOOP on the performance observed by a single query.

Backtracking: Figure 12 shows the relationship between improvement factor and the depth of backtracking in a shared cluster setting with different resource profiles. We observe that the depth of backtracking (i.e. the maximum distance of the vertex in the switched-to QEP from any running/completed vertex in the current QEP) increases with the magnitude of resource change events. 5.7% of all the runs experience a backtracking to two stages deep in the past and is triggered only by "high" volatile resource profile. 85.3% of the experimental runs with "low" volatile resource profile experienced no backtracking. We observe similar results for spot market setting. Figure 13 shows the CDF of factor of improvement w.r.t. Clarinet with and without backtracking turned on for the runs of all our TPC-DS queries when run under shared cluster resource profiles. We observe that when backtracking is turned on QOOP yields higher factor of improvement as backtracking finds better QEP switches.

Overhead of QEP switch: Figure 11 shows the overheads of QOOP in the shared cluster and spot market settings. We measure overhead as the time a job spends in switching to alternate QEPs as a percentage of total job time. The overheads in the shared cluster are 0.15% higher than in the spot market setting. This is because of the higher number of overall QEP switches when a job runs in a shared cluster – also shown in Figure 9b. On the whole, however, the overhead due to QEP switching has negligible impact ($< 1\%$) on overall job performance. The overall QEP switching overhead is low as hysteresis prevents unnecessary QEP switching and the absolute number of QEP switches in a job is low – at most 3 as shown in Figures 9a and 9b.

Robustness to Error: Figure 15 shows QOOP's robustness to error in the estimates of task resource demands and durations. We introduce X% errors in our estimated task demands and durations. Specifically, we select X in [-25, 25] as suggested by prior work [35], and increase/decrease resource demands by $task_{newReq} =$



(a) Job Performance **(b)** Inter-Job Fairness **(c)** Cluster Efficiency

Figure 14: Comparison of performance, fairness, and cluster efficiency of QOOP w.r.t. existing solutions. Higher values are better for fairness, whereas the opposite is true for the rest.

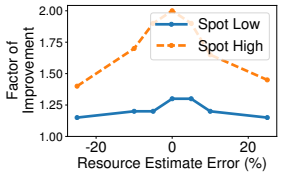


Figure 15: Error robustness in QOOP.

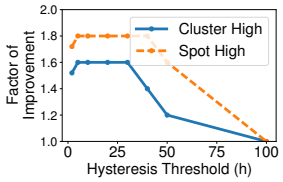


Figure 16: Effect of hysteresis on improvements.

$(1+X/100)*task_{origReq}$, and task durations change similarly. We study these errors in simulation against low and high volatile spot market resource profiles. We observe even at the highest error rates of $\pm 25\%$, QOOP offers substantial performance improvements (e.g., $1.4\times$ for the high volatile profile). For low volatile resource profile, QOOP is more robust to estimation errors: at 25% error rate, the performance improvement is $1.18\times$ compared to $1.25\times$ at no error. However, mis-estimations are costly at high volatility: errors $\geq 10\%$ cause performance improvement to drop 33% or more; nevertheless, QOOP's performance is always better than Clarinet.

Hysteresis: Figure 16 shows the effect of our hysteresis threshold (h) on the improvements. In QOOP, hysteresis prevents QEP switch unless there is an $h\%$ improvement in the estimated job completion time. We experiment with different values of h for "high" resource profiles for both spot market and shared cluster. A very high hysteresis threshold prevents switching, hurting performance. By definition, setting hysteresis parameter (h) to 0 causes more QEP switching (because of lower thresholds for QEP switching) and hence slightly higher overhead; we still see positive gains. However, for both traces, we observe that there is a wide range of h values where the factor of improvement sustains it's peak. This means that QOOP has flexibility to choose h; any value in the 10% - 25% range offers good performance at low switching overhead.

6.4 QOOP in Macro-Benchmarks

So far we have evaluated QOOP in offline, micro-benchmarks against the Clarinet QP with an aim to understand its query re-planning capabilities. In a real cluster, however, jobs arrive in an online fashion. Consequently, the impact of scheduling on job performance and its interplay with the QP become important.

In this section, we evaluate QOOP in an online setting in our shared cluster, where 200 TPC-DS jobs – randomly drawn from the 50 TPC-DS queries – arrive following a Poisson process with an average inter-arrival time of 80 seconds (Figure 14). As mentioned earlier, we compare QOOP against a wide range of solutions in both categories: scheduling and query planning. On the one hand, we consider a variety of scheduling solutions such as DRF, Tetris, SJF, and Carbyne that focus on objectives ranging from simple fairness (QOOP) to improving multiple goals. On the other hand, we consider QPs that range from static resource-agnostic planning (Volcano in Hive) to resource-aware early-binding (Clarinet) to QOOP's late-binding re-planner. Finally, in addition to focusing only on job completion time, which is useful only to individual jobs, we consider cluster-level metrics such as fairness (measured in terms of Jain's fairness index [42]) and efficiency (measured in terms of makespan).

Job Performance: First, we observe that QOOP significantly improves the average JCT w.r.t. simple state-of-the-art solutions (Tetris, DRF) and comes closest to the average JCT of SJF (Figure 14a). Furthermore, it outperforms the state-of-the-art in complex scheduling and QP alternatives: Carbyne and DRF+Clarinet, respectively. Only by combining two complicated solutions (Carbyne+Clarinet), the state-of-the-art can come close to QOOP. This suggests that the inflexibility of the current interfaces have tangible costs and overcoming them requires introducing complexities at every layer of the analytics stack.

Fairness Between Jobs: If performance were the only concern, one could get away with simply using SJF instead of using the complex alternatives or QOOP. However, performance and fairness have a strong tradeoff [35] as shown in Figure 14b – SJF has the worst fairness characteristics! We observe that while DRF and DRF+Clarinet are the most fair solutions, QOOP comes the closest to them while ensuring almost $1.5\times$ smaller average JCT.

Cluster Efficiency: Finally, Tetris performs well in its goal of packing tasks better and achieving high efficiency (Figure 14c), but QOOP again comes the closest to Tetris.

Overall, QOOP improves all three metrics – performance, fairness, and efficiency – over complex state-of-the-art solutions or combinations thereof, and achieves

these benefits using a simple scheduler with a dynamic, resource-aware QP that can re-plan queries at runtime.

7 Related Work

Other Applications: Although we focus SQL queries, the high-level principle of designing dynamic resource-aware plan switching can be applied to many other applications. This is because many frameworks use query planners to create execution plans for workloads, e.g., in machine learning [32, 44, 49], graph processing [33, 46, 48], approximation [15, 10] and streaming [60, 11, 13, 16, 50].
Query Planners in Big Data Clusters: Query planning is a well-trodden research area with numerous prior work [37]. We restrict our focus on query planners designed for distributed big data clusters that fall into two broad categories: those who plan a query in a resource-agnostic manner [3, 19] and those who are resource-aware [54]. Both, however, result in static query plans throughout the execution of a job. There is a massive body of work on adaptive query processing [26] in the context of traditional (single-machine) database systems. We focus on big data analytics in multi-node clusters.
Execution Engines: Execution engines take job DAGs and interact with the cluster scheduler to run all the tasks of each job until its completion. Examples of popular execution engines include Apache Spark [59], Dryad [39, 57], and Apache Tez [7]. Execution engines such as Tez [7] and DryadLINQ [57] allow for dynamic optimizations to the job DAG in the form of dynamism in vertex parallelism, data partitioning, and aggregation tree but lack the interfaces to make logical-level DAG switches.
Cluster Schedulers: Today's schedulers are multi-resource [30, 34, 43, 24, 17], DAG-aware [23, 34, 59], and allow a variety of constraints [61, 40, 18, 31, 58]. Given all these inputs, they optimize for objectives such as fairness [30, 41, 29, 20], performance [27], efficiency [34], or different combinations of the three [35, 36]. Over time, schedulers are becoming more complex and taking increasingly more job-level information as inputs. In contrast, we propose a simplified scheduler and argue for pushing complexity up the stack.

8 Conclusion

In this paper, we considered the problem of improving query performance in dynamic environments – e.g., in small private clusters, where resources vary with job arrivals and completions, and in clusters composed of spot instances, where resource availability changes due to changing prices. We showed that existing approaches are insufficient to adapt to dynamics because they use a fixed QEP throughout execution. We made the case for on-the-fly query re-planning and argued that it requires rethinking the division of labor among three key components of modern data analytics stacks: cluster scheduler, execution engine, and query planner. We propose a greedy re-planning algorithm, which offers provably competitive behavior, coupled with a simple cluster-wide scheduler that informs jobs of their current share. Our evaluation of a prototype using various workloads and resource profiles shows that our replanning approach driven by a simple scheduler matches or outperforms state-of-the-art solutions with complex schedulers and query planners.

Acknowledgements We thank the reviewers and our shepherd Angela Demke Brown. This work is supported by the National Science Foundation (grants CNS-1563011, CNS-1763810, CNS-1563095, CNS-1617773, and CCF-1617505), and Aditya Akella is also supported by a Google Faculty award, a gift from Huawei, and H. I. Romnes Faculty Fellowship.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] Amazon Simple Storage Service. http://aws.amazon.com/s3.

[3] Apache Calcite. http://calcite.apache.org/.

[4] Apache Hadoop. http://hadoop.apache.org.

[5] Apache Hive. http://hive.apache.org.

[6] Apache Mesos 2016 Survey Report Highlights. https://goo.gl/R6a1z2.

[7] Apache Tez. http://tez.apache.org.

[8] Google Cluster Traces. https://github.com/google/cluster-data.

[9] Hadoop Private Cluster Size Statistics. https://wiki.apache.org/hadoop/PoweredBy.

[10] Presto. https://prestodb.io.

[11] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.

[12] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[13] Trident: Stateful stream processing on Storm. http://storm.apache.org/documentation/Trident-tutorial.html.

[14] YARN Fair Scheduler. http://goo.gl/w5edEQ.

[15] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[16] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at Internet scale. *VLDB*, 2013.

[17] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[18] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[20] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.

[21] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[22] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[23] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[24] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[26] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[27] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[29] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. *SIGCOMM*, 2012.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[31] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[32] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[34] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[35] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[36] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[37] J. Hellerstein. Query optimization. In P. Bailis, J. M. Hellerstein, and M. Stonebraker, editors, *Readings in Database Systems*, chapter 7. 2017.

[38] B. Huang and J. Yang. CÜmÜlÖn-d: Data analytics in a dynamic spot market. *Proc. VLDB Endow.*, 10(8):865–876, Apr. 2017.

[39] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[40] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[41] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[42] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation, 1984.

[43] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[44] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[45] H. Liu. Cutting MapReduce cost with spot market. In *HotCloud*, 2011.

[46] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[47] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic Query Re-planning using QOOP. Technical Report TR1855, University of Wisconsin-Madison, 2018.

[48] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[49] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[50] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataow system. In *SOSP*, 2013.

[51] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[52] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a derivative IaaS cloud on the spot market. In *EuroSys*, 2015.

[53] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[54] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[55] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[56] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[57] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[58] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[59] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[60] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[61] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[62] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *SIGCOMM*, 2015.