Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud

Huangshi Tian
HKUST
htianaa@cse.ust.hk

Yunchuan Zheng
HKUST
yzhengbj@connect.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

ABSTRACT

Cluster schedulers routinely face data-parallel jobs with complex task dependencies expressed as DAGs (directed acyclic graphs). Understanding DAG structures and runtime characteristics in large production clusters hence plays a key role in scheduler design, which, however, remains an important missing piece in the literature. In this work, we present a comprehensive study of a recently released cluster trace in Alibaba. We examine the dependency structures of Alibaba jobs and find that their DAGs have *sparsely connected vertices* and can be approximately decomposed into multiple *trees with bounded depth*. We also characterize the runtime performance of DAGs and show that dependent tasks may have *significant variability* in resource usage and duration—even for recurring tasks. In both aspects, we compare the query jobs in the standard TPC benchmarks with the production DAGs and find the former *inadequately representative*. To better benchmark DAG schedulers at scale, we develop a workload generator that can faithfully synthesize task dependencies based on the production Alibaba trace. Extensive evaluations show that the synthesized DAGs have consistent statistical characteristics as the production DAGs, and the synthesized and real workloads yield similar scheduling results with various schedulers.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Cloud Resource Scheduling, Workload Analysis

ACM Reference Format:
Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362710

1 INTRODUCTION

Production data-parallel jobs increasingly have complex dependencies in computation. Modern data analytics frameworks [1, 8, 11, 31, 35, 60] compile programs into job DAGs (directed acyclic graphs) consisting of many dependent tasks. Each vertex in a DAG corresponds to a compute task, and a directed edge between two vertices (tasks) specifies their dependency. The dependency requirements of job DAGs pose significant challenges to cluster scheduling, making it difficult to balance common objectives such as high cluster utilization, fair sharing, and fast job completion [27, 28, 40].

Key to innovations in scheduler design is understanding the dependency structures of production job DAGs and their impact on scheduling. However, prior studies on cluster workloads [4, 37, 49] offer little such clue because the released production traces contain no dependency information. As DAG characterization remains lacking, many works on scheduling algorithm [9, 29, 38, 63] base their designs and evaluations on randomly generated DAGs and the limited choices of benchmark suites (e.g., TPC-DS [48] and TPC-H [47])—none of those workloads can well represent the complex dependencies of production jobs (§4-5). While some recently proposed schedulers are evaluated against real job DAGs [27, 28, 40], the production traces they used are unavailable to the public, rendering it difficult, if not impossible, to reproduce the results and promote further improvement.

In this paper, we present a panoramic view of production job DAGs through an in-depth characterization study on the Alibaba trace [3]. The trace, released in November 2018, records batch processing jobs and long-running containerized services from a cluster of 4034 machines throughout an 8-day period. We single out the dependent jobs therein and perform a comprehensive analysis on their scope of influence, dependency structures, and runtime performance. We summarize our key findings as follows.

Job DAGs are too prevalent to ignore. Our analysis shows that dependent jobs are replacing the distributed, independent jobs and becoming the majority. Almost 50% of batch jobs have dependencies, and they account for 80% of the resources consumed by all batch jobs (§3.2). The prevalence of job DAGs necessitates the dependency awareness for modern cluster schedulers.

Production DAGs are highly artificial in structure. In theory, job dependencies can form DAGs of any shape, so they are believed to be "large and complex" [28, 40]. However, we observe that the complexity of production DAGs is, to a large extent, *artificial* in that they have many distinctive features from random general DAGs. For instance, their nodes are sparsely connected, forming many small chains so that we could approximately decompose job DAGs into trees. Those trees are bounded in depth because the critical paths of job DAGs typically do not grow longer with more tasks. Instead, they become "wider" in terms of the degree of parallelism, which means more tasks can be executed in parallel. Such an artificial structure of production DAGs, if well exploited,

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362710

can greatly simplify the design of scheduling algorithms, even leading to tractable analysis under practical assumptions.

Salient variability is seen among dependent tasks. Prior workload analysis [49] reveals significant heterogeneity among production jobs, which takes a toll in the scheduling performance [5, 7, 26]. One might expect such problem to become less severe among dependent tasks within the same job because (1) upstream and downstream tasks usually run in the executors with the same configuration, and (2) sibling tasks of the same parents work on the similar intermediate data. Contrary to this expectation, the heterogeneity becomes even more pronounced to the point where dependent tasks can vary in certain metrics, such as task duration and resource usage, by more than 10^3x (§5). In comparison, a task of a data-parallel job with 8x longer duration than average is viewed as a severe straggler [5]. Therefore, high variability is turning to the "new normal" for DAG schedulers.

Standard benchmarks are not sufficiently representative of production DAGs. Existing works frequently use standard benchmarks, notably TPC-DS [48] and TPC-H [47], to evaluate the proposed cluster schedulers [27, 28, 40], so we compare their job DAGs with those in the Alibaba trace. Our comparison reveals that benchmark DAGs are neither as structurally diversified as the production DAGs, nor as comparably variable in runtime metrics. For instance, TPC-H DAGs have a rather narrow range of in-degrees and out-degrees, while TPC-DS DAGs tend to have significantly lower edge density. Such structural difference suggests that they are inadequate to evaluate a scheduler in its capability of handling complex production DAGs. We also run the two benchmarks and measure their runtime performance, e.g., task duration and resource usage. Even under the most complicated settings we could find in the literature, the dynamic range of the metrics remains an order of magnitude smaller than that of the production jobs.

As standard benchmarks may not well represent production DAGs, we design a new workload generator that can faithfully synthesize dependency DAGs from a production trace in a controllable manner (§7). Our synthesis algorithm is based on the observation that batch jobs typically have bounded critical paths as they are constrained by the program complexity. When synthesizing a DAG, we first randomly determine its critical path length based on the distribution in the trace. We then decide how nodes are distributed along the path, and generate edges to connect them. Our evaluation shows that the synthesized DAGs resemble production DAGs in terms of five metric distributions (without directly sampling from them); they also yield similar results to production traces in simulated scheduling process. Such statistical and runtime similarity attests the effectiveness of our synthesis algorithm.

We summarize our key contributions as follows.

(1) We conduct the first comprehensive analysis on the structural and runtime properties of dependent jobs at scale and identify several key characteristics of production DAGs.
(2) We compare the standard benchmarks with the production traces and discern their insufficiency in evaluating cluster schedulers.
(3) We design an algorithm that can faithfully synthesize production job DAGs in a controllable manner. We extend it to a full-fledged workload generator and release its code as open-source [55].

2 RELATED WORK AND MOTIVATION

In this section, we briefly survey existing cluster schedulers and examine their applicability in modern datacenters where dependent jobs prevail. We summarize the common assumptions held by those schedulers, and find that prior trace analyses provide little or outdated supporting information, leaving several questions for us to answer in this paper.

Dependency-Agnostic Task Scheduling Scheduling millions of jobs on tens of thousands of machines in modern datacenters poses a daunting challenge. Over the years, a myriad of cluster schedulers have been proposed, aiming at fair sharing [24, 32], low latency [25, 45], predictable performance guarantee [15, 21], fast job completion [26], straggler mitigation [50, 61], high cluster utilization [26], etc. However, most of those schedulers assume *independent* parallel tasks within a job, which is no longer the case in today's datacenters.

Job DAGs and Dependency-Aware Scheduling Data processing frameworks have evolved to a stage where computation is too complex to fit into a monolithic job, so many divide a complex job into multiple dependent tasks and organize their dependencies as a DAG. Batch analytics frameworks are the first system category to adopt such design [31, 60], followed by stream processing [8, 43] and DAG-backed SQL engines [6, 54]. The newly proposed machine learning frameworks also follow this direction [1, 11]. Dependent jobs hence have dominated production workloads.

As task dependencies gain increasing attention, many dependency-aware schedulers have been proposed recently. For example, Graphene [28] splits job DAGs and heuristically prioritizes the time-consuming and resource-intensive tasks; Carbyne [27] follows job DAGs to estimate completion time and lets jobs altruistically yield allocated resources without delaying completion; Decima [40] encodes job DAGs and schedules them with deep reinforcement learning.

We summarize common beliefs about production jobs as follows, where some (2 and 3) are based on the studies of non-DAG jobs:

(1) Job dependencies are (vaguely described as) "large and complex" [28, 40].
(2) Runtime variability is ubiquitous, e.g., straggler tasks [5], resource fragmentation [26], and data skew [7].
(3) Recurring jobs are commonplace, and their resource usage and duration are well predictable [21].

Cloud Trace Analysis Despite a body of works on cloud workload analysis [4, 14, 37, 49], the study of task dependencies remains a largely uncharted territory. The lack of such research is primarily due to (1) the lack of task dependencies, and (2) the coarse-grained workload information contained in the cluster traces. Take the Google cluster trace [49], released in 2011, as an example. The trace contains both job and task-level records, but when we attempt to infer the task dependencies from the temporal information, we find that over 98% of complex jobs are actually *parallel*. More specifically, within 98% of jobs consisting of more than five tasks, all tasks are executed in parallel, indicating no dependency between those tasks. Aside from the Google trace, workloads in the other traces, such as

Table 1: Statistics of the batch jobs in the trace. Note that we only list the attributes of our interest.

# **Jobs**: 4,201,014	# **Dependencies**: 9,449,272
... w/o Dependencies: 830,258	**Attributes of Task**
... w/ Dependencies: 3,370,756	- Start/End Time
... w/ Full Information[1]: 2,872,634	- CPU/Memory Request
... w/ over Two Tasks: 2,055,299	- # Instances
# **Tasks**: 14,295,731	**Attributes of Instance**:
... w/ Dependencies: 12,207,703	- Start/End Time
# **Instances**: 1,351,255,775	- CPU/Memory Usage
... w/ Dependencies: 1,310,672,556	- Machine ID

ATLAS [4] and Azure [14] traces, even provide no task information. The former only gives job-level traces, while the latter is collected on the VM level, where each VM can host multiple concurrent tasks. Owing to these limitations, we are unable to extract separate task information from the two traces, let alone the dependencies.

The aforementioned problems do not appear in a recent trace [37] released by Alibaba in 2017, which contains task-level information with clear signs of dependencies. Nevertheless, we choose not to reverse engineer its dependencies because the same company has released a new version of more comprehensive cluster trace [3] in November 2018 with explicit dependency information. We take it as an opportunity to answer the following questions:

(1) How complex are the job DAGs in production clusters?
(2) How do dependencies affect runtime variability?
(3) How prevalent and predictable are recurring jobs?

3 OVERVIEW OF DEPENDENT JOBS

In this section, we give an overview of the Alibaba trace (§3.1), with a focus on dependent jobs. We examine how they are distributed over time and across the cluster (§3.2) to understand their impact on the production workload.

3.1 Trace Overview

The Alibaba trace [3] we study is collected on a production cluster of 4034 machines, which records the activities of both long-running containers (for Alibaba's e-commerce business) and batch jobs across an 8-day period. In this paper, we only analyze the batch jobs and their dependencies. Readers interested in the other aspects may refer to [37] for a general analysis of a 24-hour Alibaba trace released in 2017, which contains no DAG information though.

In the trace, each batch **job** consists of one or multiple compute **tasks** which may or may not have dependencies (DAGs). A task has one or multiple **instances**[2] running the same binary but processing different data partitions. A task instance must wait until *all* instances of its *upstream tasks* complete. Table 1 gives the basic statistics about the batch jobs in the Alibaba trace. According to our contact at Alibaba, over 90% of those jobs are SQL programs supported by a Hive-like [54] framework for routine and ad-hoc data analytics. Other jobs such as Spark [6] and Flink [8] applications do exist, but only account for a small portion. Furthermore, unlike the Google workload [49], the Alibaba jobs typically have no locality preferences or placement constraints. The only common requirement is instead a (loose) completion deadline, e.g., a job generating the daily transaction summary should complete before 8 am the next day.

[1]Owing to the collection method, some jobs may have incomplete information, e.g., a few instances are not included or several attributes are missing. We sometimes exclude incomplete jobs from the analysis in this study.
[2]For those who are more familiar with Spark, the "task" in the trace is equivalent to the "stage" in Spark, and "instance" to "task".

3.2 Temporal and Spatial Distributions

To understand the impact of dependent jobs on production batch workloads, we give a holistic view of their distributions. Figure 1 shows how (DAG and all) jobs are *temporally* distributed along with their resource usage; Figure 2 illustrates how they are *spatially* distributed across the cluster, i.e., the number of machines that each job spans, and the number of jobs that each machine has run.

Dependent jobs follow (probably reversed) diurnal patterns. Batch jobs are reported to run *diurnally* in many previously released cluster traces [4]. That is, a large number of jobs run in the daytime while only a few run at night. Dependent jobs in the Alibaba trace follow a similar diurnal pattern, though in a lesser degree. Specifically, we observe 3.18x more job DAGs running in the *peak hours* than in the *slack hours*; if we expand the counting to include all batch jobs, the peak-to-trough ratio increases to 7.63x. The resource consumption, however, exhibits the opposite difference. The peak-slack variation of CPU (memory) usage is 18.94x (19.93x) for dependent jobs, and 10.67x (12.92x) for all batch workloads. That is to say, in terms of resource usage, dependent jobs demonstrate a more salient diurnal pattern than those non-DAG ones.

It is worth mentioning that such diurnal pattern may actually be *reversed*—according to our contact at Alibaba—as many batch jobs are purposely scheduled to run at night. In Alibaba clusters, batch jobs are *collocated* with long-running containerized services for improved utilization [37]. As those services are closely related to the company's e-commerce business and usually undergo peak demands during the daytime, most latency-tolerant batch jobs are scheduled at late night or early morning to avoid negatively affecting the performance of user-facing services.

Dependent jobs consume disproportionately more resources. DAG jobs compose 48.92% of batch jobs, yet they account for 77.05% of CPU cores and 80.20% of memory used by all batch workloads. In some extreme cases such as the 116$^{\text{th}}$ hour, dependent jobs, with 5.54% of population, grab 67.81% of CPUs and 81.53% of memory. On the scale of seconds, at some moment the dependent jobs occupy 98.04% of CPUs and 99.12% of memory used by all batch workloads.

Dependent jobs are evenly spread across the cluster. As shown in Figure 2 (left), over 25% of DAG jobs span hundreds or even thousands of machines, dramatically larger than the median of ten machines. Note that this could be an underestimate because the trace may miss some task instances running on another cluster. Quite different spans as those dependent job have, Figure 2 (right) suggests that they are evenly distributed across the cluster. In fact, should those jobs commonly specify *placement constraints*, which are found prevalent in the production clusters of the other companies [15, 53], we would have expected large jobs to make a fraction of machines heavily loaded, leading to spiky curves in Figure 2. As

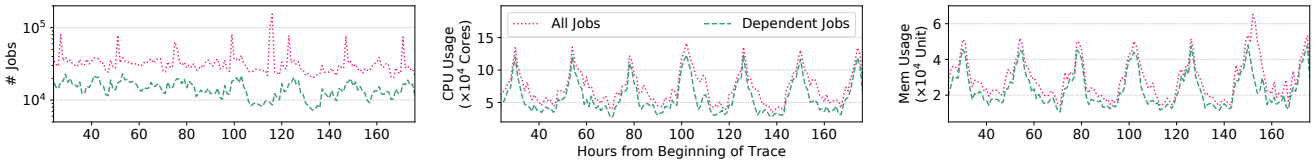

Figure 1: Hourly distributions of the number of jobs, their CPU and memory usage. The time range does not span exactly 8 days because some timestamps in the trace are misaligned [17].

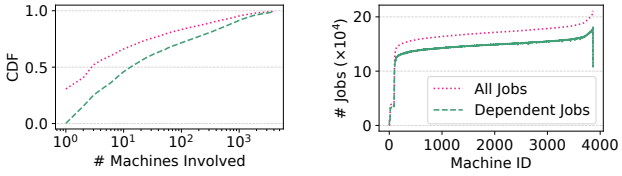

Figure 2: Distributions of the number of machines each job spans (left) and the number of jobs each machine has run (right).

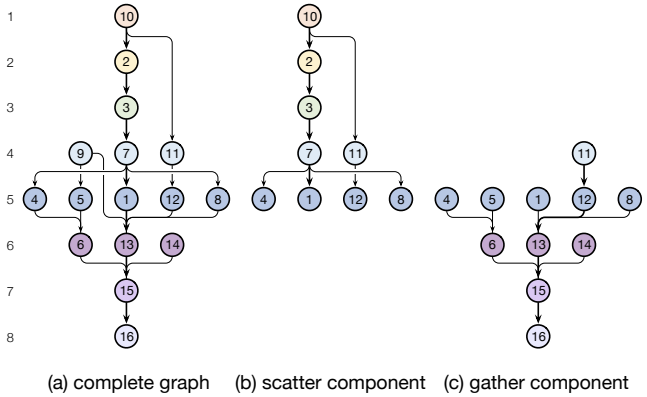

Figure 3: The decomposition of a job[3] into scatter and gather component using our heuristic algorithm. The numbers in the nodes are task identifiers.

this is not the case, we conjecture that few jobs have constraints or most constraints are not so strict. Our Alibaba contact confirms this conjecture.

4 ANATOMY OF DAG STRUCTURES

In this section, we analyze the structural properties of production job DAGs. We first show that a complex DAG can be generally decomposed into two types of trees for scatter- and gather-like operations respectively (§4.1). We then show that production DAGs are structured in a largely artificial manner (§4.2).

4.1 Dissecting Job DAGs

Data-parallel jobs frequently perform MapReduce-like operations: a dataset is first *scattered* to multiple workers and they produce intermediate data which will be eventually *gathered*. Both processes are so prevailing in the trace that most jobs can be viewed as either a scattering or gathering process, or a composite of them. To quantify their popularity, we first take them as metaphors to define two common types of jobs.

Scatter/Gather Job In the scattering process, a single task may generate data that will be consumed by multiple tasks, and those tasks may iteratively spawn more tasks. We define the jobs with such characteristic as *scatter jobs*. They feature a tree-like shape where each task, other than those in the last level, has one or more children tasks. Figure 3b gives an example scatter job. On the opposite, the gathering process reverse the shape in that the output of several tasks is collected by a single task. Such property defines the *gather jobs*, where each task, except the top-level ones, has one or more parent tasks. An example gather job is shown in Figure 3c.

These two types of jobs are so prevalent that 36.03% and 78.54% of DAGs (that contain more than 2 tasks) are scatter and gather jobs, respectively. The percentages sum up to more than 100% because our definitions allow a parent (child) to have only one child (parent) in a scatter (gather) job, meaning a *chain* of tasks can be both scatter and gather job. Such looseness in the definitions is acceptable from a theoretical perspective. As tree-structured dependency is a special case for scheduling algorithm [36], there is no reason to exclude task chains, a special case of tree, from the definition.

Decomposing Complex DAGs into Scatter/Gather Jobs Despite the prevalence of simple (scatter/gather) jobs, the trace also includes quite a few complex ones. Given that data analytics job are often composed of many primitive operators (e.g., `map` and `filter` in Spark [60]), we explore the possibility of decomposing a complex job into scatter and gather components. Algorithm 1 is designed for this purpose.[4] The key intuition behind is to prune away the nodes that are impossible to appear in a certain component. For instance, a task with two parents cannot be in a scatter component, so we remove all such tasks. After pruning, the resulting components can be viewed as a decomposition of that job, as shown in Figure 3.

After decomposing all complex jobs, we find our methods effective in that (1) the scatter and gather components overlap only to a small extent, and (2) they almost fully cover the entire DAG structures. On average, the scatter and gather components produced by our algorithms only have 11.55% tasks in common, meaning that our algorithms can effectively separate a job into multiple parts. If

[3] Its job ID is j_3985826.
[4] Owing to the approximate nature of our algorithms, we just prune tasks according to their natural order as they appear in the trace.

Algorithm 1 Find the Scatter and Gather Components

```
function FindScatter
    repeat
        prune all nodes with more than 2 parents
        prune all nodes with 0 parent and less than 2 children
    until no node is pruned
function FindGather
    repeat
        prune all nodes with more than 2 children
        prune all nodes with 0 child and less than 2 parents
    until no node is pruned
```

we reconstruct the job from its decomposition, scatter and gather components can recover 81.68% of tasks, i.e., they are included in either scatter or gather components.

The decomposition results suggest that most production DAGs are scatter or gather jobs, or can be approximately viewed as a composite of multiple simple components. Similar to the frequent communication patterns [12] stemming from primitive operators, we believe the DAG structures are also affected by them. To investigate the consequence of such effect at large, we inspect the statistical distributions of job DAGs in the coming section.

4.2 Artificiality of DAG Structure

To characterize the distinctive structural properties of production job DAGs, we compare them to standard benchmarks and randomly generated DAGs. Here we only consider the jobs *with at least 6 tasks*, because 6 is the smallest n such that, if the DAGs in the trace are randomly generated over n unlabeled vertices, the expected number of graph pairs that are *isomorphic* is less than one [51]. We introduce four datasets of DAGs in our comparison:

(1) **Alibaba** dataset where the DAGs are extracted from the trace [3]. We only include the job DAGs with complete information of tasks and instances.
(2) **Random** dataset with the DAGs generated by a uniformly random graph synthesis algorithm [34]. This dataset has exactly one-to-one relationship with the Alibaba dataset, i.e., for each DAG therein, we generate a random graph with the same number of vertices.
(3) **TPC-DS** dataset where the DAGs are generated from the namesake benchmark [48] with Spark 2.4.0 [65].
(4) **TPC-H** dataset generated similarly as above [47, 52].

Job DAGs are sparse. Our decomposition analysis (§4.1) suggests that the dependencies of many jobs actually have a tree structure. One noteworthy feature of the trees is *sparsity* because a tree with V vertices have $V-1$ edges whereas a DAG can have up to $V(V-1)/2$. To examine whether production DAGs are sparse, we define the *edge density* of a DAG as the ratio between its number of edges E and the possible maximum, i.e., $2E/V(V-1)$. Figure 5a shows the edge density of four datasets. The randomly generated DAGs have the highest edge density around 0.5, a consequence of the uniformity of the generation algorithm. In comparison, the density of production DAGs is only half of the random, hinting at lower complexity than general graphs.

Chains prevail in production DAGs. When we examine the DAGs to understand the cause of sparsity, we notice that many tasks have exactly one child and one parent, and together they form multiple *task chains*. In order to measure the prevalence of those chains, we define the *chain ratio* of a DAG as C/V, where C and V respectively denote the number of chained tasks and total tasks (vertices). Figure 5b presents the distribution of the measured chain ratios in four datasets. By a significant margin, task chains are the most prevalent in Alibaba trace, which aligns with our observation. In comparison, at least 75% of random DAGs have near-zero chain ratio, meaning that chains are rarely found in natural graphs. Therefore, we believe task chains should be treated as artifacts that deserve special attention in job scheduling.

A task can have many dependencies, but typically a few children. We next examine the in- and out-degrees of DAG vertices, where the former measures the number of dependent tasks of a given vertex (task), and the latter the number of children tasks that depend on its output. Figure 4 shows the distribution of both metrics in all four datasets. As production DAGs have sparser edges than the random ones, their in- and out-degrees are correspondingly smaller.

Specifically for in-degrees (left figure), the Alibaba curve has a long tail that stretches as far as the random curve, suggesting that a noticeable fraction of tasks depend on a significant number of others. In fact, the largest production jobs in the trace have 199 tasks and they only constitute rare cases, in which some tasks of those jobs have dependencies on *all* the other tasks.

Regarding the out-degrees, the dominating majority of the vertices in the Alibaba trace have only a few children. Specifically, 99% of vertices have out-degrees no more than 3, and 99.9% no more than 9. In comparison, 35.9% of random vertices have out-degrees larger than 3.

To sum up, most tasks in production DAGs tend to funnel multiple input datasets into some smaller output, without branching out many children tasks for parallel processing. This is in line with our prior observation that gather jobs outnumber scatter ones (§4.1). Optimizing the scheduling and execution of those gather jobs and tasks hence leads to more salient performance improvement.

Parallelism linearly increases as the task number rises. From the perspective of scheduling, the high in-degree of vertices might suggest better chances for fast job execution because, for each vertex, all its upstream tasks, if having no interdependency, can be executed in parallel. To study such parallelization opportunities, we compute the *maximum parallelism* of each job, defined as the maximum number of tasks that can be executed concurrently. For example, the job in Figure 3a has maximum parallelism of six (tasks 1, 4, 8, 9, 12 and 14). Figure 6 (left) plots the average maximum parallelism of the jobs with different number of tasks.

Notably, the maximum parallelisms of all curves other than "random" (approximately) increase linearly with respect to the task number. In fact, the ordinary linear square fitting gives a slope of 0.491, meaning every two tasks will increase the parallelism by one. However, the randomly generated DAGs have almost the same parallelism regardless of the number of tasks they have. The reason behind such discrepancy may take root in the edge density. As random DAGs have more edges (i.e., dependencies), tasks tend to

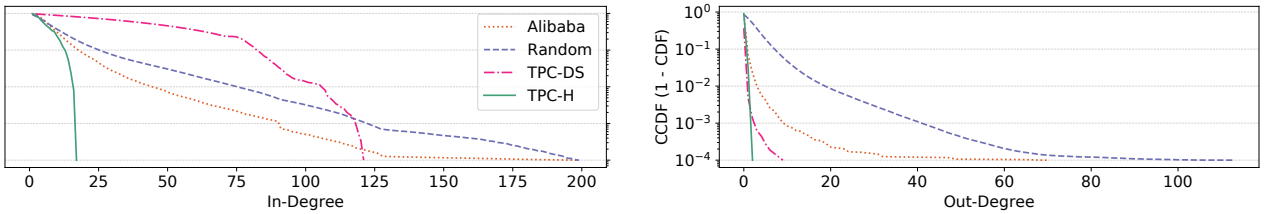

Figure 4: Adjusted[5] complementary cumulative distribution function (CCDF, the fraction that is *above* a particular value) of in-degrees and out-degrees of vertices in four datasets. For each task, its in-degree corresponds to how many dependent tasks it has; its out-degree is the number of children tasks that depend on it.

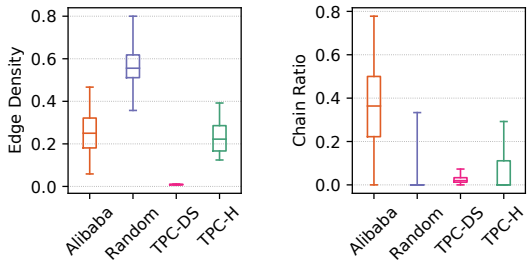

Figure 5: Edge density and chain ratio of four datasets. Each box symbol corresponds to 1st, 25th, 50th, 75th, and 99th percentiles.

be connected together, which hinders the growth of parallelism. On the contrary, production DAGs have sparser edges and thus higher parallelism.

The length of critical path stagnates when tasks increase. While parallelizing tasks shortens execution time, the minimum job completion time is dictated by the *critical path*—the longest stretch of dependent tasks in a job DAG. This motivates us to examine the critical paths of DAGs in four datasets to justify whether high parallelization helps accelerate job completion. As random DAGs do not have time information, we take the critical path length as a proxy metric. Figure 6 (right) depicts the average critical path length of the jobs with a certain number of tasks.

The random and production DAGs drastically differ in that the critical path of the former grows linearly while the latter remains stable. The slope of the random curve reaches 0.764 (computed with linear regression), meaning that, if the task dependencies are generated randomly, they almost have to be executed sequentially. In stark contrast, the production DAGs typically have quite short critical paths. Quantitatively, the DAGs in the Alibaba trace have an average critical path length of 7.68, with a standard deviation of 4.57. The result is in line with the initial purpose of many distributed frameworks: parallelizing the work for faster job completion.

[5]Notice that the y-axis is in logarithmic scale where 0 becomes an unreachable value. Therefore, we add 0.0001 to all values in order to show them completely.

Standard benchmarks do not well represent production DAGs. Our prior analysis shows that standard benchmarks fall short of structural complexity and diversity if they are to be used for evaluating cluster schedulers. On all six metrics measured above, they display a perceptible deviation from the production jobs, typically towards the simplified side. Specifically, TPC-DS has much lower edge density and smaller in-degrees, insufficient in the quantity of task dependencies. Even if some of its vertices possess reasonably large in-degrees, it still cannot cover the corner cases where a task has a huge number of upstream dependents. TPC-H has similar edge density as the production traces, but its distributions of in-degrees and out-degrees gather around near-zero area, signifying a lack of the complexity of dependencies. Figure 6 further shows that the DAGs from standard benchmarks have limited coverage of job size, only providing a small set of choices.

5 RUNTIME VARIABILITY

Now that we have characterized the structural properties of job DAGs, we proceed to analyzing their runtime performance. Prior studies [4, 49] show that cloud workloads are highly heterogeneous in that different jobs have divergent task durations, resource demands and usage. We expect such heterogeneity to become less severe between dependent tasks in the same DAG, supposedly because some tasks therein share the same intermediate data. Contrary to our expectation, however, we find the opposite to be true.

Methodology To study the impact of dependencies on runtime performance, we specify two types of dependent tasks of our interest.

(1) **Dependent Pair**: The task duo where one directly depends on the other (e.g., 1 and 7 in Figure 3a).
(2) **Dependent Set** (fork/join sets [36]): A set of (at least three) tasks that are indirectly dependent with a common parent/child (e.g., 6, 13 and 14 in Figure 3c).

Given a metric (e.g., number of instances, requested CPU cores), for a dependent pair, we calculate the *ratio* between the metric values of the two tasks as a scale-agnostic indicator of variation. For a dependent set, we first normalize the metric value of each task by that of the *smallest*. We then compute the *geometric mean* of those normalized values as the measure of variation of the dependent set. We choose geometric mean over arithmetic one because the former gives a more unbiased view of "central tendency" over

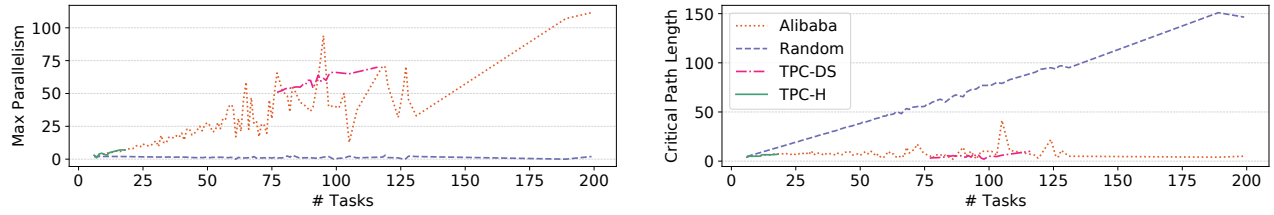

Figure 6: Average maximum parallelism and critical path length within the jobs with a certain number of tasks. The former measures the maximum number of tasks that can be executed in parallel, and the latter the longest path in the DAG.

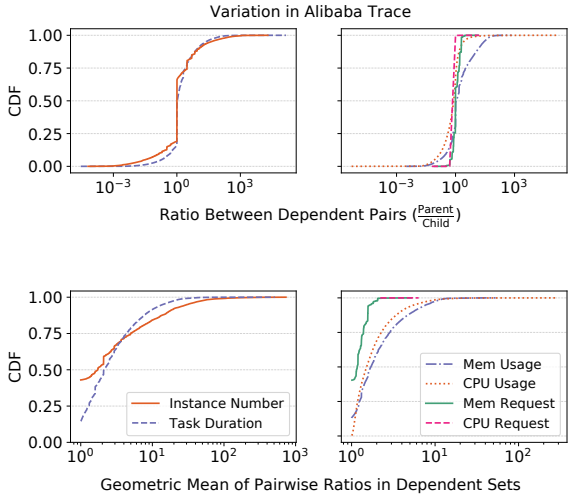

Figure 7: Variation of metrics between dependent pairs and sets. The left column shows the variation of instance numbers and task durations; the right side that of requested and used resources, i.e., CPU and memory.

skewed data [41]. Note that in our definition, the minimum possible geometric mean is 1, which corresponds to the case where tasks have the same value for a certain metric; the larger the mean is, the more diverse the values are. For the same reason as in §4.2, we limit our analysis to jobs with at least 6 tasks.

Task durations vary, sometimes dramatically. When we calculate the ratios, we notice that many tasks have zero as their durations. According to the trace description, the time unit in the trace is *second* but the system is actually running with a finer temporal scale. Hence zero actually means the task finishes within one second. Unable to recover the actual duration, we filter out all the pairs and sets where one or more tasks have zero-length durations. The left column in Figure 7 shows the extent of variation in task durations (dashed lines in the up-left and bottom-left).

In particular, among all dependent pairs, we observe 34.44% having exactly the same durations, yet 20.77% (12.83%) having variations larger than 5x (10x).[6] There even exist some extremes on five orders of magnitude. As a comparison, early work deems a task with 8x longer time as a severe straggle [5]. Aside from the numerical values, the *symmetric* shape (with respect to x=1) of the curve implies that such extent of variation exists regardless of the direction of dependencies.

The variation abates in dependent sets. While 14.57% still have identical task durations, the largest average variation plummets to $O(10^2)$, a three-order-of-magnitude decrease. We could thus infer that the task durations approximately follow a *log-normal* distribution so that the geometric mean exerts a significant averaging effect.

Instance numbers differ, sometimes more dramatically. The left side in Figure 7 also presents the variation of instance numbers (solid lines). Within the dependent pairs, although 46.67% have equally many instances, 26.46% (13.37%) of them differ in instance number by over 5x (10x). The largest variation is on the order of $O(10^4)$ and the parent is equally likely to contain more or fewer instances. As for the dependent sets, a similar 43.00% have the same instance numbers, but the largest variation still approaches $O(10^3)$, i.e., the geometric mean fails to effectively reduce the extent of variation among dependent tasks. Unlike task durations, instance numbers are dispersed in a more diverging way instead of revolving around a central value.

Resource requests are similar. Each task in the trace specifies the resource demands of each of its instances, including the number of CPU cores and the amount of memory. We plot the variation of resource demand among dependent tasks in the right column of Figure 7. The variability appears exceptionally low. Specifically, 99.87% (99.99%) of tasks pairs (sets) request for the same number of CPU cores. Though the proportion drops to 34.73% (40.43%) for the memory demand, there are still 97.61% (99.96%) of tasks pairs (sets) having less than 2x variation. The dependent pairs (sets) have their maximum variation as 16x (5.17x), relatively smaller compared with task durations and instance numbers.

We caution that such similar resource requests may be a specificity for Fuxi [64], the in-house cluster manager of Alibaba that favors jobs with low resource requests by executing them earlier.

[6] When we say "more than x times" for dependent pairs, we actually mean that the ratio is either greater than x or smaller than $1/x$ in the figure. The same rule applies to the remainder of this section.

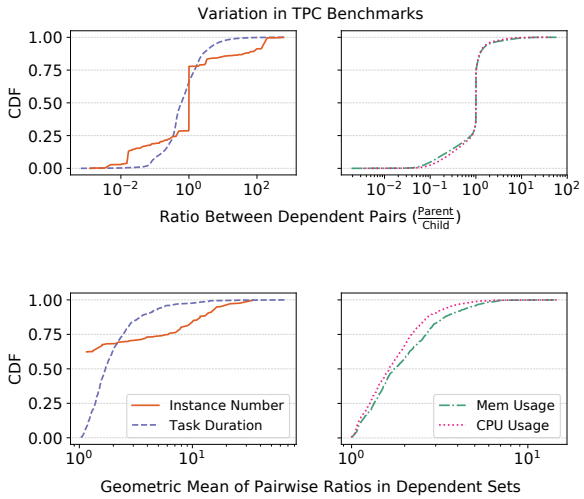

Figure 8: Variation of metrics calculated the same way as in Figure 7 but on TPC benchmarks.

Those jobs will then request incrementally for more resources during execution [37]. Therefore, the resource demand pattern we observe in the trace may not generally apply to the other cluster environments.

Resource usage varies. In order to measure the resource usage of each task, we calculate the arithmetic mean of the resource usage of all its instances. We choose not to sum them up because we have already analyzed the variation of instance numbers and we want to smooth out their impact on the resource usage. The right column in Figure 7 shows the distribution of variation among dependent tasks.

Be it within dependent pairs or sets, CPU usage varies more drastically than memory. As for memory usage, 19.78% (14.63%) of task pairs (sets) see variations less than 5%; the 90$^{\text{th}}$ percentile is 17.78x (5.48x); the largest seen variation is 703.85x. As for CPU usage, tasks with 5% of variation account for 9.59% (4.82%) of the pairs (sets); 90% of them have variation less than 6.93x (3.86x). In the extreme, the CPU usage can vary as much as $O(10^5)$ times between some dependent pair. Although each machine in the cluster only has 96 cores, 10^5 is possible because the cluster allows over-subscription [19] and fractional CPU allocation [20] (e.g., 0.05 core means allocating 5% of time of one core). For dependent sets, the maximum variation shrinks to $O(10^2)$—similarly to task durations, CPU usage can be roughly viewed as following a log-normal distribution.

Both CPU and memory show larger degree of variation during execution than in request. Sometimes the difference can be as huge as several orders of magnitude. Compared with the Google trace [49] where the most over-subscription is within 2x, the elasticity in the cluster nowadays tolerates much larger amount of resource overuse. Therefore, the resource request may not be a stable indicator of real resource usage, and their difference is far beyond the tolerance of misestimation in many schedulers [10, 27, 30, 39, 57].

Standard benchmarks have lesser variability. As standard benchmarks are frequently used in evaluating newly proposed cluster schedulers, we compare their runtime variation with the production trace. For a fair comparison, we employ the most complicated workload to our knowledge in the literature [40]. The evaluation therein mixes TPC-H benchmarks with 6 input sizes, i.e., 2, 5, 10, 20, 50, and 100 GB. We further add TPC-DS with the same input choices into the mix. Figure 8 presents the variation in those benchmarks.

Despite our efforts in complicating the workload, standard benchmarks still show much less variation than production jobs by orders of magnitude, be it in task duration or resource consumption. Therefore, even if a scheduler has been shown effective when evaluated against standard benchmarks, its performance in production environments may deteriorate because of the resource fragmentation, stragglers, and workload imbalance brought by such a high degree of variation. For instance, some poor scheduling decisions (e.g., accidentally starting a long job that blocks others) may not cause too much harm when tasks have comparable durations, but the variation in production tasks may magnify its negative effect.

6 JOB RECURRENCE

Prior work [21, 27, 33] reports that most production jobs in Microsoft's clusters are *recurrent*, and their resource usage can be accurately inferred from past runs for better scheduling. We thus analyze the Alibaba trace from this perspective to validate the generality of that finding.

Methodology From the trace, we cannot tell directly if two jobs running at different times are recurrent, as no meta information about the jobs (e.g., application programs and business units) is provided. We therefore turn to *structural isomorphism* and *temporal periodicity* as the two indicators of recurrence. The former is frequently exploited in scheduler design [21, 27, 30]; the latter is allegedly prevalent among production jobs [33].

We first group together the jobs with isomorphic dependency structures (§6.1). Within a group, we pick out periodic jobs and treat them as recurrent (§6.2). We then analyze their predictability in terms of runtime and resource usage (§6.3). Note that we only include jobs composed of at least 6 tasks in the analysis, because those with fewer tasks may coincidentally have the same DAG structure as stated in §4.2.

6.1 Isomorphic Jobs

If two jobs have isomorphic DAGs (structurally similar), they have a high chance of having the same processing logic. Since it is NP-hard to check graph isomorphism, we turn to an approximate approach by drawing DAGs using a layout algorithm [22] and comparing the output images. If the result images are identical, then both DAGs must be isomorphic. Note that the opposite is, however, not true, so our results below just give an *underestimate*.

The vast majority of job DAGs are repetitive. We find that 99.52% of job DAGs have at least one structurally isomorphic counterpart in the trace; 90.19% have at least 13; 75.01% have at least 117. These numbers suggest that DAGs are not uniformly distributed in the graph space, which is in line with our decomposition result (§4.1) in that most jobs are composed of several artificial patterns

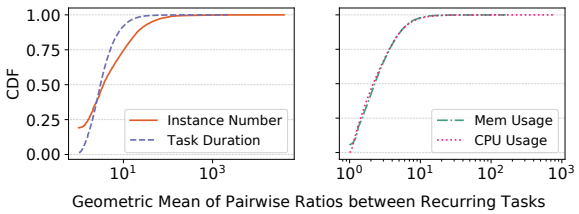

Figure 9: Variation among "recurring" tasks computed with the same method as in Figure 7.

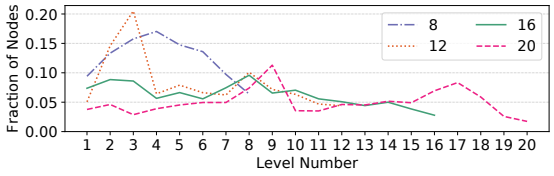

Figure 10: Node distributions in terms of level in jobs with critical path length of 8, 12, 16 and 20, respectively.

(e.g., gather or scatter components). However, according to [33], it is estimated that only 60% of production jobs are recurrent, so we have to heighten the selection standard to identify them.

6.2 Periodic Jobs

As recurring jobs are usually scheduled to run periodically, we use the periodicity as a selection criteria to shortlist candidates among isomorphic jobs. To obtain the scheduling interval of two jobs, we calculate the difference between the start times of their earliest tasks. If a set of jobs have roughly (i.e., within 5% deviation) the same scheduling intervals in-between, we view them as recurrent. Lacking in the clue of real periods, we expediently choose 15 minutes, 1 hour, and 1 day—three common periods in production jobs reported by Microsoft [33].

The isomorphic jobs periodically run in those three intervals respectively account for 3.27%, 4.02%, and 6.60% of all jobs, much fewer than those reported by Microsoft [33]. Nevertheless, we believe that they are inevitably underestimated because (1) the Alibaba trace does not include the exact job submission time, and (2) the jobs have no strict SLOs and may undergo an indefinite delay.

6.3 Predictability of Recurring Jobs

We now study if the resource usage of recurring jobs can be well predicted from the past runs. Within those periodic jobs, we single out those that have exactly the same resource request and treat them as "recurring" jobs. We don't require the instance numbers to be same because they may be data-dependent. For instance, a Spark application will spawn more instances when data partitions grow. Among those "recurring" jobs, we match their tasks and compute the inter-task variation using the same method as in §5. Note that such matching between two isomorphic job DAGs is not unique (e.g., consider two symmetric graphs), so we choose the one with minimal variation. Concretely, we sort the metrics and follow their order to match tasks sequentially. The computed variation is shown in Figure 9.[7]

"Recurring" tasks may have high runtime variability. As shown in Figure 9, between "recurring" tasks the instance numbers can vary as much as 10^4x, and CPU usage 10^3x. In fact, 69.25% of "recurring" tasks have larger than 2x variation in instance number, 75.69% in task duration, 54.15% in CPU usage, and 57.61% in memory usage. Such a large variation poses non-trivial challenges to the common approach of predicting task attributes (e.g., duration, resource usage) based on their previous execution [10, 27, 30, 39, 57]: to our knowledge, most predictive schedulers assume no more than 100% of prediction error in evaluations [30].

[7] Our Alibaba contact comments that the extremely high repetitiveness of DAGs conforms with their impression on recurring jobs [18]. Considering such prevalence, we believe at least a considerable proportion of selected tasks are indeed recurrent.

7 DEPENDENCY SYNTHESIS

Having uncovered several statistical peculiarities of production DAGs, we switch to a generative perspective by studying how to *authentically synthesize dependency graphs*. Faced with such requirement are the researchers who want to evaluate their proposed scheduling algorithms against (synthesized) production DAGs—standard benchmark suites fall short in this purpose (§4.2 and §5). With synthesized dependencies, we could extend many previously released traces [4, 49] that include no dependency information and take them to benchmark newly proposed DAG schedulers.

Based on our observations (§7.1) about the critical paths in production jobs and how tasks are distributed along them, we design an algorithm (§7.2) to randomly generate a dependency graph. We evaluate (§7.3) its output in terms of both the statistical distribution and dynamic scheduling. Finally, we elaborate on how we extend it into a full-fledged cluster trace generator (§7.4).

7.1 Observations on DAG Structures

As seen in §4.2, the critical path length of job DAGs remains stable regardless of the number of tasks. Such stability motivates us to closely examine how tasks are spread along the path. Specifically, for each task (vertex) v, we assign it a *level number* $l(v)$. We say a level assignment of a DAG is *valid* if $l(c) > l(p)$ holds for any tasks c and p, where c (child task) depends on p (parent task). That is, children tasks have larger level numbers than their parents. As an example, the left-sided numbers in Figure 3a constitute a valid level assignment. Among many valid level assignments, we are interested in the *compact* ones where the sum of the differences between all parents and children are *minimum*, i.e., minimize $\sum_{p,c} l(c) - l(p)$. Referring back to Figure 3a, the level assignment is compact, but it is not the only choice because we could elevate task 11 to level 2 while maintaining the compactness. We design an algorithm to obtain a level assignment for each DAG, whose details can be found in our technical report [56]. In general, it recursively assigns the number from the root nodes and then compacts the number backwards whenever possible.

Cross-level edges are rare. In the trace, we observe that most edges only span two adjacent levels, i.e., the level difference between

Algorithm 2 Synthesize a DAG with a Given Size

– s: size of DAG, i.e., number of vertices
– CP: random generator of critical path length given a DAG size
– LV: random generator of level number given a critical path length

1: **function** SynthesizeDAG(s)
2: Initialize N as an array of s vertices.
3: $len \leftarrow CP(s)$
4: **for** $n \in$ a random permutation of N **do**
5: **if** n is in the first len vertices **then**
6: Set $n.level$ as its index in the sequence.
7: **else**
8: $n.level \leftarrow LV(len)$
9: **for** $l \leftarrow 1, \cdots, (len - 1)$ **do**
10: **for** $n \in$ vertices on level l **do**
11: $r \leftarrow \lceil \#\text{ vertices on level } l+1 / \#\text{ vertices on level } l \rceil$
12: $n.children \leftarrow$ sample r vertices from level $l + 1$
13: **return** N

two endpoints of an edge is one, the minimum possible value in a valid assignment. To have a comprehensive view, we calculate the level difference of all edges in the Alibaba trace and 1000 randomly generated DAGs. Noticeably, 96.54% of edges in production DAGs stay between two adjacent levels, as opposed to 29.92% in random DAGs. The result means the output of many production tasks is often only consumed by another task immediately after it, and there rarely are subsequent tasks that further depend on it. Such characteristic makes it feasible for us to only consider two adjacent levels when synthesizing the edges.

Nodes are not evenly distributed on each level. Since most tasks only depend on those exactly one level above them, we count the tasks on each level and inspect how they are distributed. Figure 10 plots the distributions of the jobs with critical path length of 8, 12, 16, 20, respectively. Production jobs show considerable variability in node distribution. For instance, many 8-node-long DAGs have "thick waists" where quite a few tasks amass around the middle levels; 12-node-long DAGs tend to be "heavy-headed" by having many tasks in the first three level; 16- and 20-node long DAGs have roughly bimodal distributions with the modes in different places. Such irregularity renders futile our attempts to theoretically model the distribution and hints at the necessity of direct sampling.

7.2 Synthesis Algorithm

Based on the observations above, we propose Algorithm 2 for DAG synthesis, which is underpinned by two pre-defined random generators. The first is *critical path length generator CP* that takes the vertex number as its input. For vertex number no greater than 35, we direct count the occurrences of length in all jobs with that size and randomly draw one following the proportional probabilities. For the jobs with more than 35 tasks, we aggregate them and draw the path length from their overall distribution. We differentiate between those two cases because not all job sizes exist in the trace, especially when they are relatively large. Hence we have to interpolate the distribution in those gaps. We choose 35 as the threshold because it is the length of the longest critical path. The second generator *LV* is used for assigning a *level number* to each vertex. It takes the critical path length as the input and randomly output

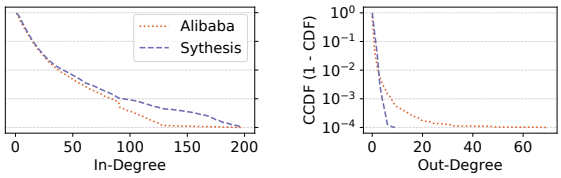

Figure 11: The distributions of in- and out-degrees in original and synthesized DAGs.

a level number based on the level distribution in all jobs with the given length.

Algorithm 2 can be viewed as a three-step process. (1) We determine the critical path length for a DAG (Line 3). It comes as the first step because critical path length is a defining difference between real and random jobs. Production jobs and standard benchmarks, despite their disparity in many other topological metrics, have similar distribution of critical path length (Figure 6b), only differing in the range. Thus we take it as the basis of the synthesis. (2) To decide how vertices are distributed along the critical path, Line 4 through 8 assign the level number to each vertex. The first several vertices are numbered sequentially to ensure that all levels have at least one vertex. (3) Finally, Line 9 to 12 generate edges between each two adjacent levels. The step tries to make all vertices connected to at least one next-level vertex because the production DAGs have all their vertices connected. Here we only construct edges between two levels because of the rarity of cross-level edges in the trace (§7.1).

7.3 Evaluation

To evaluate our algorithm, we raise two crucial questions:

(1) Do synthesized DAGs have similar structural characteristics to production DAGs?
(2) Do they yield similar results during scheduling?

Static Properties Section 4.2 has shown that production DAGs have special distributions in various metrics, so we examine whether our synthesis algorithm is able to capture them. To keep the calculation consistent, we synthesize, for each DAG in the Alibaba trace, a DAG with the same number of vertices. Then we compute the distributions of the previously used metrics and plot them in Figure 11 and 12. The synthesized and production DAGs are closely similar in the majority of metrics. Although the distributions of in- and out-degrees appear pictorially dissimilar, they succeed in capturing the key characteristics of production trace. The maximum out-degree almost reaches 200, covering the extreme case in the trace; the in-degrees are mostly small, reflecting that most tasks have few children.

Dynamic Properties We further simulate the scheduling process on both production and synthesized traces. We sampled an hour-long trace from the original one. In our simulator, we set the machine configuration the same as the Alibaba servers (96 units of CPU, 100 units of memory [3]) and choose such a cluster size

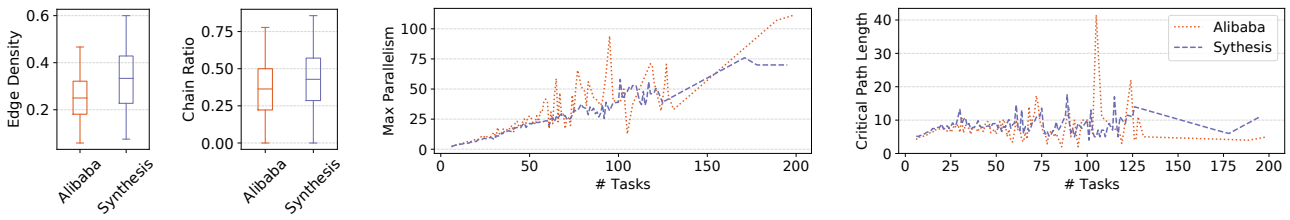

Figure 12: Comparison of chain ratio, edge density, maximal parallelism and critical length between production and synthesized DAGs.

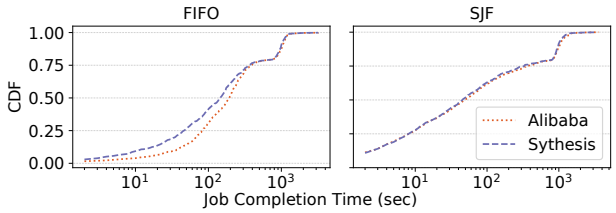

Figure 13: We simulate scheduling on both production and synthesized traces. The figures show the distributions of job completion time using FIFO and SJF schedulers, respectively.

that the average load of bottlenecked resource reaches 0.8. We consider two widely used scheduling policies, First-In-First-Out (FIFO) and Shortest-Job-First (SJF). After simulating the scheduling of production jobs, we replace their dependencies with our synthesized DAGs and repeat the experiment. For each simulation, we compute the CDF of job completion time and depict them in Figure 13. Under both scheduling policies, the distributions are almost the same, indicating that our algorithm manages to capture the runtime characteristics of production DAGs.

7.4 Trace Generation

We further extend the synthesis algorithm to a full-fledged cluster trace generator whose output trace can be controlled in several aspects. This will be useful when the trace is not directly usable. Supposing a user is evaluating a scheduler against a cluster with less resources than Alibaba cloud, it will be problematic to directly replay the trace because of some overly high resource demand. Also, the user may want to evaluate with workload of different load or heterogeneity, so it is desirable to have some control knobs for trace generation.

Our generation tool offers to users three tunable parameters. (1) They could set the size the resource amount of the target cluster with **cluster configuration**. We then rescale the resource usage of tasks to make them fit into the target. (2) If the users want to increase or decrease the cluster load, they could specify the **load** parameter. For lower load, we simply down-sample the trace; for higher, we up-sample jobs and replace the dependency with synthesized one to ensure diversity. (3) In stress testing, the users can make the jobs vary to a larger extent by setting **heterogeneity amplifier**. We further scale up the extreme metrics to the user-defined ratio. With those parameters, users can experiment with the trace in a more customized and flexible manner. The tool has been released as open-source [55].

8 DISCUSSION

In this section, we discuss the implications of our measurement analyses to the scheduler design. As our measurements are limited to a single source of trace collected in an Alibaba cluster, we make a caveat to the generality of the conclusions drawn from our study.

8.1 Implications to Scheduler Design

The prevalence of job DAGs underscores the necessity of having **dependency-aware schedulers**. In the Alibaba trace, DAGs compose nearly half of the job's population, account for 85% of tasks, and grab over 80% of CPUs and memory (§3.2). Some large DAGs may even span over 1000 machines. By all means, job DAGs have already become the *first-class citizens* in the cloud. We thus anticipate dependency awareness to be *indispensable* for scheduler design.

Though dependency DAG complicates the scheduling problem, there is a silver lining that schedulers may not need to deal with the full complexity posed by general dependency graphs. As our analysis repeatedly demonstrates, the DAGs of production applications have characteristics far from the random general graphs (§4.2). To name a few, their in-degrees are mostly bounded; adding more tasks tends to increase the parallelism rather than expanding the critical path; the dependency edges seldom cross levels, i.e., connecting two faraway tasks in terms of dependencies. Our simple, yet faithful synthesis algorithm also serves as an evidence that the DAGs are highly artificial (§7). Therefore, although many graph-related problems are NP-hard, we believe DAG scheduling stands a good chance to be tackled by **theoretical approaches**, such as approximation, randomization algorithms with provable performance bound under some practical assumptions.

Aside from DAG awareness, the runtime variability of production jobs necessitates the **robustness to variation and misestimation** in scheduling algorithms. Many state-of-the-art schedulers [10, 27, 30, 39, 57] make scheduling decisions based on the estimation of job durations and resource usage. Many of those schedulers are robust against 25% error in estimation, though the most tolerant stretching to 100%. However, our analysis has shown that the claimed resource demand may differ from the actual usage by orders of magnitude. Such a large discrepancy poses serious

challenges to the predictive scheduling algorithms. We have seen promising solutions offered by *non-clairvoyant* schedulers (e.g., Kairos [16], Aalo [13], NC-DRF [58]) which make scheduling decisions without the exact job information. Another prospective solution goes to adaptive scheduling in that the scheduler quickly reacts to the environmental changes. Quintessential examples include RoPE [2] and QOOP [39] that can dynamically adjust the execution plan with runtime information.

Finally, when it comes to scheduler evaluations, there is a **fundamental limit of using standard benchmarks** which are structurally deviant from the production DAGs and far less variable in runtime performance. This signifies the necessity of using production traces for a comprehensive evaluation of cluster schedulers.

8.2 Limitations

We have to acknowledge that, throughout the entire study, our analysis is limited to a single source of trace collected from an Alibaba cluster. Though we have attempted to include more production traces in our analysis, we were unable to extract the required information from the existing public traces (§2). Therefore, it remains unproven whether the conclusions we draw from the Alibaba workload generalize to the other production clusters. Below we discuss the potential limitations of our analysis.

Business Domain The source of this trace, Alibaba incorporation, is an e-commerce company, whose core business includes online shop hosting, item recommendation, and online transactions. Other applications, such as social media and search engines, may have different web characteristics, which can lead to the disparity in many technical aspects, e.g., job types, framework choices, stored data, etc. For instance, indexing web pages [46] is a unique workload to search engines; the various content types supported by social media result in an overflow of BLOB data [42]. As a result, the structural and the runtime characteristics we observed in the Alibaba trace may not hold in the traces from other companies due to the different mix of job and data types.

Target Applications As pointed out in §3.1, the current trace mainly consists of SQL programs that are based on a distributed engine similar to Pig [44], Hive [54] and Spark SQL [6]. Since SQL has a specific syntax, the underlying generated jobs are likely to share some common patterns. For instance, a typical `SELECT-FROM-WHERE` statement will be translated into a dispersion of a dataset followed by an aggregation. Such patterns could result in some special properties of task dependencies which, however, may not apply to the other applications, such as distributed machine learning [62], deep learning [59], and long-running applications [23]. For the clusters dominated by those types of workloads, our conclusions, especially the structural characteristics, may not directly apply.

Scheduling Infrastructure Alibaba has developed an in-house scheduling system, Fuxi [64], with several specialized design that may contribute to certain runtime characteristics of the trace. For instance, it supports fractional resource usage, a potential cause of the exceptionally high runtime variability, while other schedulers may not face the same problem. Also, as mentioned in §5, Fuxi's preference to lower resource requests and its incremental resource allocation would encourage many jobs to make more conservative resource demands. Unlike Fuxi, other schedulers may need to handle a greater variety of requests than those in the Alibaba trace.

9 CONCLUSION

In this analysis, we aimed at understanding the characteristics of data-parallel jobs with inter-task dependencies at cloud scale. By analyzing the Alibaba trace, we have uncovered several unique structural characteristics of production DAGs in Alibaba, including sparse edges, small out-degrees and the critical paths with a relatively stable length. The simpler structures than the general graphs hint at the potential tractability of theoretical approaches. Yet the runtime variability among dependent tasks becomes no lesser, but actually aggravated. Irrespective of directly or indirectly dependent tasks, they show variation of runtime metrics on several orders of magnitude. The variability persists even among "recurring" tasks, which necessitates the robustness and adaptiveness in future scheduler design. During the analysis, we have also found the standard TPC benchmarks insufficiently representative of production workload, so we have designed a workload generator that can faithfully synthesize task dependencies in the Alibaba trace in a controlled manner. Our evaluation demonstrates that, compared with the production DAGs, our synthesized graphs have almost identical statistical distributions and produce similar scheduling results with various scheduling algorithms.

ACKNOWLEDGMENTS

We thank our shepherd, Timothy Zhu, and the anonymous reviewers for their valuable feedbacks that help improve the quality of this work. We are deeply grateful to Haiyang Ding and Yihui Feng for providing detailed background information about the Alibaba trace and offering insightful comments to an early version of this paper. This research is supported by RGC ECS grant under contract 26213818. Huangshi Tian was supported in part by the Hong Kong PhD Fellowship Scheme. Yunchuan Zheng was supported in part by the Huawei PhD Fellowship Scheme.

REFERENCES

[1] Martín Abadi, Paul Barham, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*.
[2] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Reoptimizing data parallel computing. In *NSDI*.
[3] Alibaba. 2019. Alibaba Cluster Trace Program. https://bit.ly/2K8DWCa
[4] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the diversity of cluster workloads and its impact on research results. In *ATC*.
[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: Attack of the clones. In *NSDI*.
[6] Michael Armbrust, Reynold S Xin, Cheng Lian, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*.
[7] Laurent Bindschaedler, Jasmina Malicevic, et al. 2018. Rock You Like a Hurricane: Taming Skew in Large Scale Analytics. In *EuroSys*.
[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2015).
[9] Chen Chen, Wei Wang, and Bo Li. 2018. Performance-Aware Fair Scheduling: Exploiting Demand Elasticity of Data Analytics Jobs. In *Proc. IEEE INFOCOM*.
[10] Chen Chen, Wei Wang, Shengkai Zhang, and Bo Li. 2017. Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees. In *IEEE Conference on Computer Communications (INFOCOM)*.

[11] Tianqi Chen, Mu Li, et al. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Neural Information Processing Systems, Workshop on Machine Learning Systems* (2015).

[12] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A networking abstraction for cluster applications. In *HotNets*.

[13] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*.

[14] Eli Cortez, Anand Bonde, Alexandre Muzio, et al. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*.

[15] Carlo Curino, Djellel E Difallah, Chris Douglas, et al. 2014. Reservation-based scheduling: If you're late don't blame us!. In *SoCC*.

[16] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive data center scheduling without runtime estimates. In *SoCC*.

[17] Haiyang Ding. 2019. Is there some problem with the time_stamp in v2018?? https://github.com/alibaba/clusterdata/issues/52

[18] Haiyang Ding. 2019. Private Communication. Online Meeting.

[19] Haiyang Ding. 2019. Question about CPU allocation on containerized online service. https://github.com/alibaba/clusterdata/issues/19

[20] Haiyang Ding. 2019. Question Regarding Normalized Memory Usage. https://github.com/alibaba/clusterdata/issues/61

[21] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*.

[22] Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and K-P Vo. 1993. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering* (1993).

[23] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: scheduling of long running applications in shared production clusters. In *EuroSys*.

[24] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *NSDI*.

[25] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *OSDI*.

[26] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. (2015).

[27] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters.. In *OSDI*.

[28] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*.

[29] Zhiming Hu, James Tu, and Baochun Li. 2019. Spear: Optimized Dependency-Aware Task Scheduling with Deep Reinforcement Learning. In *Proc. IEEE ICDCS*.

[30] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. 2015. Scheduling jobs across geo-distributed datacenters. In *SoCC*.

[31] Michael Isard, Mihai Budiu, et al. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*. ACM.

[32] Michael Isard, Vijayan Prabhakaran, Jon Currey, et al. 2009. Quincy: fair scheduling for distributed computing clusters. In *SOSP*.

[33] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, et al. 2016. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*.

[34] Jack Kuipers and Giusi Moffa. 2015. Uniform random generation of large acyclic digraphs. *Statistics and Computing* (2015).

[35] Sanjeev Kulkarni, Nikunj Bhagat, et al. 2015. Twitter heron: Stream processing at scale. In *SIGMOD*.

[36] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)* (1999).

[37] Qixiao Liu and Zhibin Yu. 2018. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: a View from Alibaba Trace. In *SoCC*.

[38] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2019. Online Job Scheduling with Resource Packing on a Cluster of Heterogeneous Servers. In *Proc. IEEE INFOCOM*.

[39] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. 2018. Dynamic Query Re-Planning using QOOP. In *OSDI*.

[40] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2018. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963* (2018).

[41] Donald McAlister. 1879. The law of the geometric mean. *Proceedings of the Royal Society of London* (1879).

[42] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, and other. 2014. f4: Facebook's Warm BLOB Storage System. In *OSDI*.

[43] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*.

[44] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*.

[45] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *SOSP*.

[46] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*.

[47] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* (2000).

[48] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. Tpc-ds, taking decision support benchmarking to the next level. In *SIGMOD*.

[49] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*.

[50] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *SIGCOMM*.

[51] Robert W Robinson. 1977. Counting unlabeled acyclic digraphs. In *Combinatorial mathematics V*. Springer.

[52] Savvas Savvides. 2018. tpch-spark. https://github.com/ssavvides/tpch-spark

[53] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. 2011. Modeling and synthesizing task placement constraints in Google compute clusters. In *SoCC*.

[54] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, et al. 2009. Hive: A Warehousing Solution over a Map-reduce Framework. *VLDB* (2009).

[55] Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. Alibaba DAG Trace Generator. https://github.com/All-less/trace-generator.

[56] Huangshi Tian, Yunchuan Zheng, and Wei Wang. 2019. *Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud.* Technical Report. HKUST. https://www.cse.ust.hk/~weiwa/papers/huangshi-socc19-techreport.pdf

[57] Alexey Tumanov, Timothy Zhu, Jun Woo Park, et al. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*.

[58] Luping Wang and Wei Wang. 2018. Fair coflow scheduling without prior knowledge. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*.

[59] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.

[61] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *OSDI*.

[62] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. Slaq: quality-driven scheduling for distributed machine learning. In *SoCC*.

[63] Xiaoda Zhang, Zhuzhong Qian, Sheng Zhang, Xiangbo Li, Xiaoliang Wang, and Sanglu Lu. 2018. COBRA: Toward Provably Efficient Semi-Clairvoyant Scheduling in Data Analytics Systems. In *Proc. IEEE INFOCOM*.

[64] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *VLDB* (2014).

[65] Yunchuan Zheng. 2019. TPC-DS on Spark. https://github.com/SimonZYC/tpcds-spark.