

Unearthing inter-job dependencies for better cluster scheduling

Andrew Chung, *Carnegie Mellon University;* Subru Krishnan, Konstantinos Karanasos, and Carlo Curino, *Microsoft;* Gregory R. Ganger, *Carnegie Mellon University*

https://www.usenix.org/conference/osdi20/presentation/chung

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

Unearthing inter-job dependencies for better cluster scheduling

Andrew Chung* Subru Krishnan† Konstantinos Karanasos† Carlo Curino† Gregory R. Ganger*
*Carnegie Mellon University †Microsoft

Abstract

Inter-job dependencies pervade shared data analytics infrastructures (so-called "data lakes"), as jobs read output files written by previous jobs, yet are often invisible to current cluster schedulers. Jobs are submitted one-by-one, without indicating dependencies, and the scheduler considers them independently based on priority, fairness, etc. This paper analyzes hidden inter-job dependencies in a 50k+ node analytics cluster at Microsoft, based on job and data provenance logs, finding that nearly 80% of all jobs depend on at least one other job. Yet, even in a business-critical setting, we see jobs that fail because they depend on not-yet-completed jobs, jobs that depend on jobs of lower priority, and other difficulties with hidden inter-job dependencies.

The Wing dependency profiler analyzes job and data provenance logs to find hidden inter-job dependencies, characterizes them, and provides improved guidance to a cluster scheduler. Specifically, for the 68% of jobs (in the analyzed data lake) that exhibit their dependencies in a recurring fashion, Wing predicts the impact of a pending job on subsequent jobs and user downloads, and uses that information to refine valuation of that job by the scheduler. In simulations driven by real job logs, we find that a traditional YARN scheduler that uses Wing-provided valuations in place of user-specified priorities extracts more value (in terms of successful dependent jobs and user downloads) from a heavily-loaded cluster. By relying completely on Wing for guidance, YARN can achieve nearly 100% of value at constrained cluster capacities, almost 2× that achieved by using the user-provided job priorities.

1 Introduction

Data lakes have become core elements of modern data-driven enterprises, providing required data storage and analysis infrastructure (see Fig. 1). Data lakes enhance data processing via a combination of two critical properties: (i) a highly consolidated, multi-tenant infrastructure that enables multiple teams of data scientists and engineers to share resources rather than each having their own, and (ii) low data access barriers that allow easy data sharing between users and various types of data analytics applications. Combined, these properties increase data re-use [4, 27] and reduce overall computational resource-hours consumed [31, 33].

This same data and resource sharing creates a new challenge: hidden inter-job dependencies. We say that Job 2 depends on Job 1 if Job 2 takes as input any output file generated and stored into the shared distributed file system by Job 1.[1] For example, in Fig. 1, Job 3 (from Org 3) depends on Job 2, which in turn depends on Job 1. We refer to these as *hidden* dependencies, to contrast them with explicit computation DAGs managed by schedulers within workflow managers [30,40,41], because there is no indication of such dependencies indicated in the job submissions—the dependencies are not expressed to the cluster scheduler.

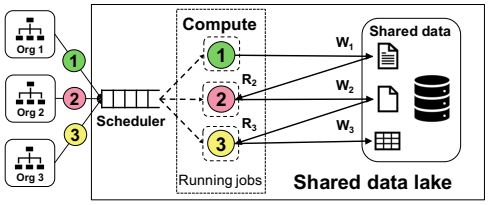


Figure 1: Data lake overview. Different jobs submitted by different organizations share the same compute infrastructure and read (R) and write (W) to the same storage system, thereby creating *inter-job dependencies* as jobs consume the output of other jobs. e.g., Job 2 (from Org 2) reads a file written by Job 1, so Job 2 depends on Job 1.

The advent of GDPR [56] forced large companies such as Microsoft to invest in infrastructures to track data provenance and data movement both within the data lake and to external components. This created an unprecedented opportunity to uncover and exploit these inter-job dependencies for scheduling: We analyze data extracted from petabytes of job and data provenance logs for 90 days of a 50k+ server cluster (part of Microsoft's Cosmos data lake [6, 12]) shared by over 1300 users from more than 150 internal organizations. In total, our analysis covers over 4 million submitted jobs and 16 million inter-job dependencies. We find that almost 80% of submitted jobs depend on output generated by at least one other job. Indicating the breadth of sharing, many dependencies are cross-organization, with 20% of jobs depending on jobs submitted by another organization.

Despite so much inter-job dependence, systems provide little support for addressing associated challenges. For example, in Cosmos, different users and organizations make their own decisions regarding when to submit jobs and how to set job priorities. Ideally, all co-dependent organizations and users would set up clear Service Level Agreements among themselves to ensure timely arrival of input data for business-critical analyses. Yet, we see signs of insufficient coordination

[1] Our nomenclature and analyses focus on fundamental dataflow dependencies among batch analytics jobs, not distributed stream processing or artificial inter-relationships caused by resource contention.

to ensure that jobs' outputs are produced in time for consumption by dependent jobs. For example, 13% of submitted jobs depend on output files from jobs that execute at a lower priority, which can result in priority inversion since job schedulers are not dependency-aware. More broadly, 34% of recurring jobs are submitted without checking if inputs they depend on are available, failing immediately if they are not.

The Wing dependency profiler efficiently processes prior job and provenance data to predict the impact of each new job on future jobs and user downloads. Although it is inherently difficult to know what future jobs will depend on the output generated by a current job, Wing finds success by focusing on recurrence. Previous workload studies have shown that > 60% of jobs in data analytics environments are *recurrent* and suggest that dependencies of these jobs can similarly follow certain patterns [34, 51]. Our analyses in Cosmos confirm that inter-job dependencies are recurrent (79% of all inter-job dependencies are recurrent), with jobs of the same template exhibiting recurring input consumption patterns. As such, Wing uses *historically recurring dependencies* to (i) analyze and predict relationships between common, dependent recurring jobs, and (ii) guide a cluster scheduler to value jobs in a way that accounts for hidden dependencies.

To explore Wing's efficacy, we pair Wing with stock YARN scheduling (*Wing-Agg*), replacing user-provided priorities with Wing-guided priorities. Specifically, we use number of downloads attained associated with a job's outputs as an approximation for job value,[2] and assign priorities to jobs based on *value efficiency* [8, 28, 44] (job-value divided by resource-time-used). We use trace-driven simulation to evaluate Wing-Agg, compared to using the user-provided priorities (as used in Cosmos), when the goal is to maximize the overall value attained. We find that Wing-guided scheduling achieves up to 66% more value than the Cosmos default, under cluster capacity crunch. Further, when organizational cluster resource boundaries are removed, a Wing-guided scheduler can achieve nearly 100% of value at constrained cluster capacities, almost 2× the value achieved by scheduling based on user-provided job priorities.

Contributions. This paper makes three primary contributions: (i) It presents the first detailed public study of hidden inter-job dependencies in a large-scale data analytics cluster, revealing important problems and opportunities; (ii) it describes a novel system for extracting historical inter-job dependencies from provenance data, at scale, and predicting the impact of a newly-submitted job on future jobs and users; (iii) it shows that use of such predictions can allow a modern scheduler, with minimal changes, to better serve the overall workload by prioritizing the highest-impact jobs.

[2] While job output download-counts are imperfect as ground-truth for job value, a limited check (§4.3) against known important levels for six business-critical jobs indicates that it at least sometimes behaves reasonably.

2 Hidden inter-job dependencies in Cosmos

This section describes and analyzes hidden inter-job dependencies in a large production data lake (Cosmos), highlighting observations that affect resource scheduling decisions and opportunities. It provides an overview of Cosmos and inter-job dependencies, introduces terminology used through the rest of the paper, and quantifies the prevalence and characteristics of hidden inter-job dependencies.

2.1 Cosmos

Overview. Cosmos is one of the largest big data analytics infrastructures in the world. Deployed internally within Microsoft, it is made up of multiple clusters, each with 50k+ nodes [12]. Within Cosmos, more than 80% of infrastructure capacity is dedicated to SCOPE jobs [6, 12], which are batch data analytics jobs similar in nature to Apache Spark [57] and MapReduce [14]. Our work primarily focuses on SCOPE jobs and inter-job dependencies between them.

CosmosFS and operations. SCOPE jobs submitted to a Cosmos cluster read input from and write output to a distributed file system known as the *CosmosFS*. A user can also access CosmosFS through a front-end service to upload or download files directly. We call actions performed on files in CosmosFS, either by SCOPE jobs or through the front-end, *operations*.

Continuous logging. Cosmos continuously tracks and logs data provenance and job telemetry (e.g., compute-hours, submission/completion time, and job structure metadata) into external services: *ProvRepo* stores data provenance and and *JobRepo* stores job telemetry. Our analyses and Wing use these logs to figure out inter-job dependencies.

Job template vs job. A job template [32, 34] is a program to be executed (one or multiple times) in Cosmos, while a job is an actual execution of a job template. Each submission of a job template results in a job.

SCOPE job submission patterns. Common patterns used to submit SCOPE jobs within Microsoft include:

(i) *Manual submissions*: Where a job is manually submitted.

(ii) *Workflow managers*: Workflow managers allow users to automate SCOPE job submissions using *workflows*. Workflows consist of inter-dependent jobs that often map to a business task, and can be triggered periodically or conditionally. Within Microsoft, there are at least five major production workflow managers, each with thousands of users.

(iii) *Custom shell scripts*: Scripts can be set up to perform automated job submissions for users. This method is more flexible, but requires specialized management.

2.2 Inter-job dependencies

How are inter-job dependencies formed? We say that a job *A* depends on a job *B* if *A* consumes *any* of *B*'s output as input. As a concrete example of a recurring cross-organization inter-job dependency, periodic jobs deployed by the data compliance team process CosmosFS access logs, which are generated hourly by the CosmosFS team, to detect data compliance

Characteristic	Description	Heuristic
Recurring	*Recurring jobs* are jobs whose template is submitted many times over time, often to analyze fresh data. *Recurring dependencies* are dependencies occurring between jobs of two recurringly-submitted job templates.	Borrowing from Morpheus [34], jobs are identified as *recurring* if (a) jobs of a template are submitted at least three times over a period of three months, with at least one submission each month, (b) templatized job names are an exact match, and (c) source-code signatures are an approximate match. Dependencies are identified as recurring if both the upstream and the downstream jobs are recurring.
Ad-hoc	*Ad-hoc jobs/dependencies* are those not recurring.	*Ad-hoc jobs/dependencies* are those not identified as recurring.
Periodic jobs	*Periodic jobs* are recurring jobs that are submitted "on-the-clock" at a fixed cadence (e.g., submitted every hour at the start of the hour).	Jobs of a template are identified as *periodic* if they are recurring and if job submissions have near-constant inter-arrival time. To determine if inter-arrival times are near-constant, we use the coefficient of variation (CV). Jobs with small CV in their inter-arrival times are identified as *periodic*, while others are *aperiodic*.
Polling	Jobs are *polling* if they scan and wait for their inputs to become available before their submission. Input dependencies of polling jobs are similarly polling.	Jobs are identified as *polling* if they (a) are not identified as periodic, indicating that they are not submitted on a clock, (b) never fail due to missing files from their recurring upstream jobs, and if (c) they are submitted within 15 minutes of the completion of their latest-completing dependent job. Input dependencies of a polling job are *polling*.
Hard dependencies	Dependencies are *hard* if the downstream job requires the output(s) of the upstream job to be able to run successfully. If the input(s) of the downstream job is not ready by the time of its submission, the downstream job fails with a missing file exception.	Dependencies are identified as *hard* if they are (a) ad-hoc, (b) recurring *and* > 95% of jobs of the same template consume the output of only one job of the same upstream job template, or (c) if the downstream job consumes the output of the same number of upstream jobs of the same job template all the time, indicating that they expect the same number of inputs from the same number of jobs from the upstream template.

Table 1: Summary of and heuristics to identify and characterize job and dependency types.

issues. There are many ways inter-job dependencies can form, and while some inter-job dependencies form through careful negotiation between users/organizations, most are formed *organically*, such as via:

(i) *Data discovery through data catalogs*: A user finds an interesting dataset while browsing through Microsoft's internal data catalog, and sets up a job to analyze the dataset.
(ii) *Script inheritance*: A user wanting to submit a SCOPE job to analyze a popular dataset often starts with a script written and shared by others, that contains logic to extract the dataset. The new script, while containing custom logic, often retains parts of the original script (e.g., priority settings).
(iii) *Logically related intra-workflow jobs*: Workflows, which can consist of multiple inter-connected jobs, are often constructed to improve job modularity and manageability. Each run of a workflow potentially creates many inter-job dependencies, as jobs within a workflow are inter-dependent. Note that, although a workflow manager may know about these inter-job dependencies, there is no interface for a workflow manager to express them to Cosmos.

Characteristics of jobs and dependencies. Our analyses uncovered a few major types of dependency and job characteristics based on job submission patterns (Table 1). The three most important job and inter-job dependency characteristics for our purposes are *recurring*, *ad-hoc*, and *hard*.

Challenges. Among the many ways in which inter-job dependencies can form and evolve, most promote loosely maintained (or non-existent) contracts between inter-dependent jobs in favor of developer convenience. This leads to an environment in which most users know little about upstream jobs that produce their input datasets, and even less about downstream jobs that depend on the data their jobs produce. These sub-optimal inter-job dependency configurations are often only exposed as a result of capacity impairment, unexpected job failures, or data/job audits. Indeed, inter-job dependencies are *hidden* through the availability of the many disaggregated solutions to manage and submit jobs and workflows, prompting us to develop Wing to uncover these dependencies.

2.3 Observations on inter-job dependencies

This section motivates our work on exploiting inter-job dependencies by describing consequential empirical observations about our inter-job dependency data, observed over three months in a single Cosmos cluster.

Observation 1 (Recurring jobs & dependencies): Most jobs and dependencies are recurring. Recurring jobs make up 68% of all submitted jobs (the other 32% of jobs are ad-hoc), while recurring dependencies make up 79% of all dependencies (the other 21% of dependencies are ad-hoc). Recurringness of jobs and dependencies suggest predictability, which we show to be achievable in §3.

Observation 2 (Priority mis-configurations): In Cosmos, jobs are assigned resources in declining priority order, where the priority of a job is assigned by the job's submitter. Here, we find that potential priority mis-configurations are frequent within Cosmos: jobs of 21% of job templates have the chance to be systematically priority-inverted—i.e., recurring jobs consuming their output have a higher priority. In addition, up to 33% of ad-hoc jobs are assigned higher priority than the average recurring job submitted within the same hierarchical queue,[3] where recurring jobs are often production jobs [34].

Observation 3 (Uncoordinated jobs): Many jobs are submitted without explicit coordination with respect to the completion of their upstream jobs—i.e., these jobs do not wait for their input to become available nor are tolerant to missing input, yet they are submitted blind with respect to the avail-

[3] *Hierarchical queues* designate resource shares of an organization in clusters at Microsoft. Priorities are only comparable between jobs in the same queue.

ability of their inputs. Such jobs make up 34% of recurring jobs, and can be susceptible to failure due to missing input from an upstream job not completing in time.

Observation 4 (Cross-org jobs & dependencies): Cross-org jobs and dependencies are common at Microsoft. Up to 95% of organizations have cross-org dependencies. Of all dependencies, 33% are cross-org, and 17% of template dependencies are cross-org, where a *template dependency* is a dependency between recurring jobs of two job templates. Furthermore, 28% of jobs and 23% of recurring jobs are involved in cross-org dependencies. Cross-org dependencies can be harder to manage because they require coordination between jobs across hierarchical queues and between job owners across different organizations.

Observation 5 (Jobs are highly inter-connected): Modeling jobs and their dependencies as a directed acyclic graph (DAG), where inter-job dependencies represent edges, we find that more than 50% of jobs are inter-connected in a single weakly connected component (CC), and CCs of sizes ≥ 10 cover more than 80% of all jobs. We also find that the larger a CC, the more bottom-heavy it is—the failure of certain jobs in such large CCs can cause significant amounts of cascading failure downstream.

Observation 6 (Many jobs can be load-shifted in time): Analyzing when the outputs of jobs are consumed by both downstream jobs and users, we find significant opportunity to delay, or load-shift jobs in time, which allows cluster operators to mitigate capacity crunches or reduce power cost [2, 3, 36]. A job has the potential to be able to be load-shifted by up to T hours if it is (1) recurring, (2) has a gap of $> T$ hours before its output(s) are consumed, *and* (3) has run times of $< T$ hours. (1) allows the job to be identifiable in the future, and (2) and (3) ensure that the job has enough slack to be safely delayed by up to T hours. We find that 31, 27, 22, and 14% of all jobs can be potentially load-shifted by up to T = 1, 3, 6, and 12 hours, respectively. [4]

Discussion. We have seen failure due to lacking input and priority inversions happen during manual inspection of job logs and dependency graphs, but we can not provide counts. We have also seen that: (1) users can and do fix their jobs, sometimes at the cost of sub-optimal performance and results, to work around issues, such as by by consuming stale data; and (2) some of these problematic inter-job dependencies can be masked with sufficiently available resources. A better understanding of inter-job dependencies can help us uncover problematic mis-configurations before they show up.

3 Inter-job dependency predictability

Inter-job dependencies show potential in guiding scheduling; but it is unrealistic to expect job submitters to provide all inter-job dependencies up-front due to the fragmented nature of inter-dependency knowledge (§2.2). While *inter-job dependency recurrence* shows promise, for Wing to effectively guide schedulers with inter-job dependencies, recurring inter-job dependencies also need to be *predictable*—i.e., it is important that past dependencies tell us something about the future. In this section, we use a simple model to predict future occurrences of recurring inter-job dependencies, and show that inter-job dependencies can be predictable.

3.1 Prediction model

Given a specific point in time where a job j_u of template J_u ($j_u \in J_u$, where the symbol "∈" is used as shorthand for "of instance") has arrived, for each recurring job template J_d that depends on the output of template J_u in a recurring fashion, our prediction model has two targets: **(T1)** *whether or not* a recurring job $\in J_d$ will arrive and depend on j_u in the future and **(T2)** *when* the first instance of such a job will arrive.

Model for (T1): Will a downstream recurring job arrive? For (T1), our model uses a configurable *prediction threshold* $tr\%$ ranging from 0 to 100 to predict whether or not a job $\in J_d$ will arrive: If $\geq tr\%$ of prior jobs $\in J_u$ have their outputs consumed by a job $\in J_d$, then the predictor predicts `true`; otherwise it predicts `false`.

Model for (T2): When will a downstream job arrive? For (T2), our model aggregates previously observed recurring dependencies where the upstream job $\in J_u$ and the downstream job $\in J_d$, and computes the median elapsed time from the *submission* of the upstream job to the submission of the first dependent downstream job.

3.2 Predictability evaluation

Dependencies change slowly over time. Dependency patterns of recurring jobs change slowly over time, and making predictions based on inter-job dependencies over longer periods of time presents challenges. For example, in (T1), using a month of inter-job dependency data to train our model to predict the arrival of dependent jobs occurring in the next month only allows us to capture at most 77% of upcoming jobs. Regularly training our model on a week of data to predict for the next week works comparatively well, because (1) it allows us to capture up to 95% of upcoming jobs and (2) it allows us to characterize the dependencies of 89% of job templates (covering 97% of all jobs), since jobs of most templates are submitted with an inter-arrival time of less than a week (with daily submissions being the most common).

(T1) metrics and model performance. We evaluate the prediction quality of our model on (T1) based on precision[5] and recall.[6] Fig. 2 examines the tradeoff between precision and

[4] While load-shifting is an interesting topic for future research, we do not directly address methods for load-shifting in this paper.

[5] *Precision* is defined as the number of true positives (TPs) divided by the sum of TPs and false positives. Precision can be thought of as the percentage of positive predictions our model makes (i.e., a downstream job will arrive) that are truly relevant (i.e., such a downstream job actually arrives).

[6] *Recall* is defined as TPs divided by the sum of TPs and false negatives. Recall can be thought of as the percentage of relevant results that our model is able to correctly predict.

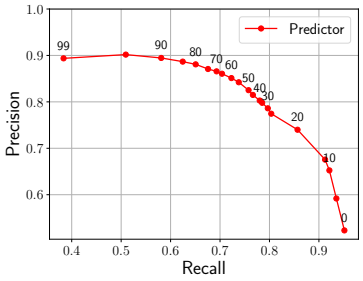


Figure 2: (T1) precision-recall tradeoff. *Predictor* shows the precision-recall tradeoff our dependency-based job arrival predictor makes. Each point on the curve specifies a different setting for the prediction threshold (tr). As $tr \rightarrow 100\%$ (more selective), a larger fraction of predictions are relevant (more precision), but less relevant jobs are captured in total (less recall).

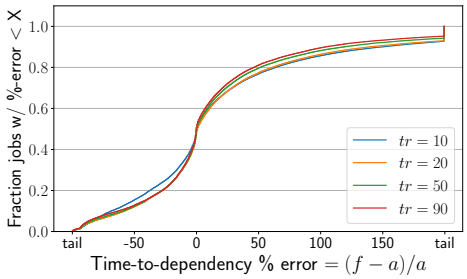


Figure 3: (T2) Time-to-dependency (TTD) prediction. This figure shows our predictor's performance on predicting TTD from the submission time of the upstream job, at different settings of tr in a CDF. f is the *forecasted* TTD, and a is the *actual* TTD. While being more precise ($tr \rightarrow 100$) does not yield better TTD predictions, it does affect predictions on whether or not a job will arrive.

recall for our model using various settings for the prediction threshold tr. As the model becomes more selective with respect to which downstream jobs will arrive ($tr \rightarrow 100\%$), it retains less relevant dependencies in total, but the dependent recurring jobs it predicts to arrive mostly do show up. The reverse is true as the model becomes less selective ($tr \rightarrow 0\%$).

We discuss the evaluation of our model based on a threshold that balances precision and recall. A common way to identify such a threhold is to select the threshold that maximizes *precision* $*$ *recall*. We find that $tr = 20\%$ yields the greatest *precision* $*$ *recall*, and therefore evaluate our model by setting $tr = 20\%$. The threshold used in an online prediction service can similarly be tuned from week-to-week based on observed precision and recall, though the specific target to optimize depends on the penalties associated with making mistakes in recall or precision.

(T2) metrics and model performance. To evaluate the performance of our model on predicting *when* a downstream job $j_d \in J_d$ will arrive at the arrival time of an upstream job j_u, j_d must satisfy two conditions: our model must predict j_d to arrive based on jobs that have already arrived during a point in time in the execution trace *and* it must actually arrive. Our evaluation focuses on jobs that satisfy both above conditions.

To evaluate the performance of our model for (T2), we use the *Root Mean Squared Error (RMSE)* and the *Median Absolute Error (MAE)* metrics to measure prediction error in absolute time units. RMSE measures error by computing the root of the average of squares of errors, while MAE measures error by computing the median of absolute error = $|forecast - actual|$, over all predictions. To measure relative error, we use the *percentage error* metric: it computes $(forecast - actual)/actual$ for each prediction.

While we discuss the evaluation of our model on (T2) setting $tr = 20\%$, we find that confidence in job arrival prediction only slightly affects time-to-dependency prediction quality. This does not mean that the setting of tr is inconsequential, as tr affects the predictions of whether or not a job will arrive. Here, we evaluate the time-to-dependency predictions only for jobs that are both predicted to arrive *and* actually arrive.

We observe the RMSE and MAE of our model to be 2.5 hours and 22 minutes, respectively: MAE is smaller, as RMSE can be skewed by large mis-predictions at the tail. While our absolute errors can be improved using more sophisticated techniques, we find that our model predictions are reasonable for most jobs in our workload in terms of relative error, as shown in Fig. 3 in the form of a cumulative distribution function (CDF), for different settings of tr: the arrival of 50% of arrived jobs $\in J_d$ are predicted within $\pm 20\%$ of its actual arrival. But, there is also a non-trivial number of significant over-estimates: the arrival of 7% of arrived jobs $\in J_d$ are over-estimated by 2× or more—i.e., the actual jobs arrive more than 2× earlier than predicted. While this may not explain all mis-estimations, we have found that aperiodic recurring jobs (such as those that are manually triggered) and jobs that depend on the outputs of multiple jobs are prone to greater mis-estimates (our simple model presented here only tries to predict the arrival of a future job based on one of its directly upstream recurring jobs).

4 The Wing dependency profiler

This section describes Wing, an end-to-end dependency profiler meant to be run intermittently (e.g., weekly) that uncovers historical, hidden inter-job dependencies from data provenance logs. It performs a series of analyses using these inter-job dependencies in-tandem with historical job telemetry, yielding characterizations of jobs and inter-job dependencies such as signs of misconfigured priorities between recurringly-dependent jobs (§2.3), predictability of upcoming jobs (§3), and estimates of recurring jobs' aggregate value considering their impact on downstream jobs that rely on their outputs, directly or indirectly (§4.3). These characterizations are ultimately used to inform better scheduling decisions, where its benefits are explored in more detail in §7.

4.1 Architecture

First, we introduce related systems and data sources upon which Wing depends, and provide an overview of Wing's architecture, shown in Fig. 4.

Input data sources. Wing relies on the following data sources, from which we derive job dependencies and insights thereof: (i) *JobRepo* preserves job telemetry (e.g., compute-hours, submission/completion time, and job structure metadata) for submitted jobs. Wing uses JobRepo to derive recurring jobs and their historic statistics. (ii) *ProvRepo* tracks data provenance across Microsoft to support auditing and compliance applications [56]. Specifically, it stores data provenance across systems deployed within Microsoft, including but not limited to Cosmos. ProvRepo is used by Wing to uncover historic inter-job dependencies, and from there, infer recurring dependencies between recurring jobs.

Analysis pipeline. Wing's data analysis pipeline, primarily composed of a workflow of inter-dependent SCOPE jobs, is managed by a workflow manager and is periodically executed in Cosmos. The pipeline reads data from JobRepo and ProvRepo and writes its output to be consumed by WingStore.

WingStore. The *WingStore* is a service that hosts the resulting analyses of Wing's analysis pipeline, periodically renewed each time a new instance of the pipeline completes. Given a historical job or the identifier of a recurring job, one can look up relevant historical job and inter-job dependency data: Such historical job data include, but are not limited to, distributions of job runtime and compute-time used. Historical inter-job data include distributions of job fan-in/fan-out, recurring inter-job dependencies, and distributions of number of downstream jobs. The WingStore is the interface between Wing's analysis and a Wing-guided resource manager.

4.2 The Wing pipeline: Single-hop analysis

Wing considers both *single-hop* and *multi-hop* dependencies in its analyses. The former occur when jobs directly consume the output(s) of another job. The latter are indirect dependencies between jobs that are connected by means of intermediate jobs. Here, we focus on the derivation of single-hop dependencies, which multi-hop dependencies are built upon, from historical provenance data stored in ProvRepo.

Single-hop dependency derivation. To derive single-hop dependencies from provenance data in ProvRepo (stored roughly in the form of <`input`, `operation`, `output`>, but with much more detailed context), we perform a self-join on the ProvRepo dataset with the condition of p_1`.input = `p_2`.output`. A näive self-join across multiple months of data is extremely compute intensive and can yield incorrect results, as a single file can be written multiple times by different jobs. To reduce join complexity and ensure correctness, we apply the following additional rules on the join:

(1) *R/W correctness*: The read must occur after the write. i.e., p_1`.operation` must occur after p_2`.operation`.
(2) *Last-writer wins*: If multiple writes occur on a single file, the read only depends on the latest write prior to the read.
(3) *Time windowing*: The time between the read and the write operations are at most T days, where we set $T = 30$.[7]

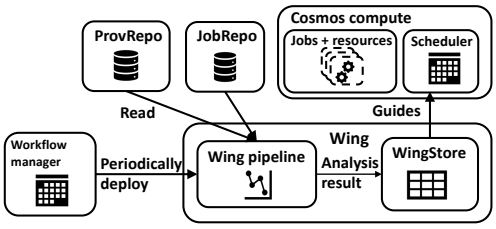


Figure 4: Wing architecture. A workflow manager periodically submits Wing's pipeline to Cosmos. Upon pipeline completion, results of its analyses are loaded in to WingStore, which informs Wing-guided schedulers (§6.2) with job and dependency characteristics.

Time windowing can reduce join complexity and allow our analyses to account for inter-job dependencies more *fairly*—if time windows are not applied over an observation period, operations issued earlier necessarily have a higher chance to be depended-upon. In other words, for each operation between days 31–60[8] in our dataset, time windows give them equal opportunity (in wall-clock time) to be depended-upon by directly-dependent operations.

Heuristics to identify recurring jobs & dependencies. A key to the analyses that we perform is the identification of recurring jobs, for which we employ the time-tested heuristic proposed in Morpheus [34] and applied in multiple production environments [34, 51]. Through the identification of recurring jobs and uncovered single-hop dependencies, the Wing pipeline further derives recurring dependencies and uncovers dependency characteristics of jobs using similar heuristics, described in Table 1. While ideally, we would like the full semantics of how inter-job dependencies are formed, due to the availability of the many different ways to submit a job (§2.1), our usage of heuristics is necessary. Sampling 25 jobs for manual verification, we confirm that our heuristics categorize jobs and dependencies correctly for 24 of the jobs.

4.3 Motivating multi-hop analysis: Job valuation using aggregate downloads

Companies can benefit more from their infrastructure investment through effective scheduling that prioritizes the completion of the most valuable jobs. But, often times, inter-job dependencies have not been considered when evaluating the importance of jobs—e.g., a job with high value can potentially depend on jobs with low value. In these cases, inter-job dependency awareness is key to ensure that upstream jobs do not disrupt high-value downstream jobs. Here, we look at why inter-job dependency analyses beyond direct dependencies (i.e., *multi-hop analyses*) can inform better, dependency-aware

[7] In retrospect, we should have set $T = 31$ to capture all monthly cycles, but our results based on $T = 30$ remain valid because (1) 98% of dependencies occur within a week, and (2) jobs of 89% of templates (97% of all jobs) have mean inter-arrival times of less than a week.

[8] Operations between days 31–60 are analyzed because we observe fully over time windows of 30 days both operations they depend on (days 1–30) and those that depend on them (days 61-90).

valuation of jobs to improve scheduling, and explore using the *number of downloads attained* associated with the outputs of a completed job as a proxy-metric for job value.

Priority assignments. To prioritize jobs today, schedulers in most production data analytics environments, including in Cosmos, use priority assignments to determine a job's order in its claim to resources. In this context, the notion of job value is often translated into a priority assignment on the job—the greater a job's value, the higher its priority. However, priorities in clusters are difficult to set correctly (Observation 2), and even at Microsoft, whose multi-billion dollar clusters are carefully provisioned and whose user-base is highly skilled, incidents triggered by late completion of hand-picked, closely monitored, and highly valued production jobs still occur due to mis-configured priorities.

Multi-hop value impact. The completion of a job can often be associated with some measurement of monetary value to a company. For example, jobs computing Bing's search indices directly impact the revenue of Microsoft. We term the direct value associated with the completion of a job its *job-local* value. However, the delay or failure of a job may not only affect its users and consumers of its output: through analyses of Cosmos's job DAG (Observations 3 and 5), we find that the delay or failure of certain jobs impact a lot more jobs and users than others. Hitches in the execution of these jobs are likely to cause much more financial and operational damage to users and organizations within the company due to the ripple effects they can create downstream, yet their impact might not always be obvious. While prior work [18, 34] suggest that finishing jobs prior to the arrival of their first directly-dependent job is important, quantifying the *aggregate value* of a job necessitates inter-job dependency analyses extending beyond a single hop (i.e., *multi-hop analyses*). Fig. 5 showcases a toy example that computes such an aggregate value for the root job of a dependency tree.

Approximating value impact with agg. downloads. Although determining the true dollar-value of jobs is difficult, we find it promising to evaluate the importance of jobs based on their historical *aggregate user downloads*, which measures hypothetically if a job fails, how many download operations it will affect (directly or indirectly) in total. In developing Wing, we have also experimented with several alternative metrics e.g., sum of cpu-hours and number of downstream jobs. Number of downloads was preferred by our resource management team because file downloads (1) are the most direct way users interact with a job's output; (2) can be easily interpreted and understood; and (3) because file downloads can be used to quantify how soon the output(s) of a job are used upon its completion. The properties of file downloads allow aggregate downloads counts to provide a proxy-measure to how the delayed or failed outputs of jobs can impact users in and out of Microsoft. Aggregate download counts also implicitly capture the number of downstream jobs that can be impacted by the failure of a job through their associated output downloads.

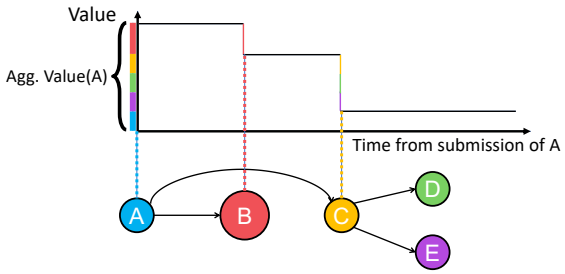


Figure 5: Value aggregation and value decay. In this toy example, jobs A–E are submitted at strict, absolute times, where the x-axis denotes time relative to the submission of job A. B and C have hard dependencies on A, and D and E have hard dependencies on C. The aggregate value of A is the sum of the aggregate values of B and C and A's own job-local value. With Wing, we can model how the aggregate value of A decays as it fails to complete by the time its downstream jobs arrive, losing the value of B at the time of B's submission, and collectively losing the values of C, D, and E at the time of C's submission (D and E depend *indirectly* on A through C, so if C fails, D and E will also fail). In this example, A retains its job-local value until the end.

While further work is required to confirm that aggregate download counts represents job value and to explore how it should be combined with other signals (e.g., user-provided priorities), we use it in this paper as our approximation of value.

Sanity-checking aggregate downloads as job value. We conducted a sanity check, using aggregate download counts for job valuation to see how it matches up with pre-existing notions of job importance. To that end, we obtained a list of six recurring job templates hand-curated by the Cosmos resource management team at Microsoft, each vetted to be significantly important to Microsoft's operation. We then look at Wing's ranks of those jobs.

Our results show that our valuation scheme mostly holds up for the most important jobs: We find that jobs of five of the six templates are consistently ranked by our scheme to be among the top 4% of all jobs submitted, with jobs of one template still ranking in the top 11%. We also measure relative rankings by user-specified priority and by our heuristic among jobs submitted to the same organizational queue, since priorities are only relevant when compared to other jobs sharing the same queue. For four out of the six hand-curated job templates, Our heuristic produces organizationally-relative rankings within 5% of priority assignment rankings. For one of the six job templates, our valuation scheme produces a ranking lower than that produced by priority assignments by up to 11%. For the last of the six job templates, however, we produce a ranking *higher* than that produced by priority assignments by 50%. This is surprising because we expected priority assignments for these six job templates, which are all verified to be highly important, to be extremely well-tuned, with highly-ranked priorities assigned to jobs of all six templates. Yet, jobs of the last template are only ranked at the 49^{th}

percentile of all submitted jobs within its queue by priority assignment—this mis-configuration may lead to significant issues once the queue becomes more heavily-loaded.

Future work: Further validating agg. downloads as value. We acknowledge that accurate job valuation is a difficult problem that requires further study, and that different companies can have different notions of job value. While further efforts are ongoing at Microsoft to validate the efficacy of our job valuation scheme (e.g., conducting surveys of Cosmos users), Cosmos's resource management team has noted that our valuation scheme is better than any of their existing heuristics used for job valuation, and are considering adopting it to aid in rolling out job upgrades and using it as a weighting function to report certain cluster performance indicators (e.g., reliability).

4.4 The Wing pipeline: Multi-hop analyses

Wing provides a flexible iterative solution implemented on top of SCOPE for performing *downstream multi-hop analyses*, in which for a given job, we analyze properties of its directly and indirectly-dependent jobs. Provided a set of single-hop inter-job dependencies, our framework allows the computation of both the transitive closure and aggregate statistics of all sub-DAGs rooted at each job in an inter-job dependency DAG (defined in Observation 5). Such multi-hop analyses are important to effectively guide scheduling decisions, as it can compactly characterize each job's downstream impact: i.e., if a job fails or is delayed, how will its downstream jobs and users be affected (§4.3)? Our framework generalizes the algorithm proposed in Owl [9], which allows multi-hop dependency analysis to be applied to other applications, e.g., fixing priority inversions[9] for Cosmos jobs.

Algorithm input. Our algorithm input is a single-hop job dependency DAG specified as a relational table, where the first column (`job`) holds the dependent job and the second column (`depOn`) holds the depended-upon job.

Algorithm output. Our algorithm outputs a relational table describing multi-hop dependencies. The first column (`job`) holds the downstream job, the second column (`depOn`) holds the (potentially multi-hop) upstream job, and the third column (`agg`) holds Wing-computed weights aggregated along all paths between the pair of up/downstream jobs.

Aggregation Functions (AFs). Each downstream multi-hop analysis specifies the following *Aggregation Functions* (AFs):

- *Weight function* (`wt_fn`): `wt_fn` takes in a job and its in- (or out-) edges as input, and outputs a weight `wt` for each graph edge. This operation is done once to convert the input DAG into an edge-weighted DAG.
- *Edge operation* (`e_op`): For two vertices t and v connected by an intermediate vertex u, `e_op` performs an aggregation of weights between a pair of (potentially auxiliary) in- (t,u) and out-edges (u,v) of u, constructing a new auxiliary weighted edge connecting t and v. Specifically, it computes the weight for an auxiliary edge based on new edges explored in each iteration between two indirectly connected jobs. This operation should be *distributive* over the `p_op` (defined following).
- *Path operation* (`p_op`): `p_op` aggregates weights on all explored paths between two jobs. While a unique path cannot be explored multiple times, the algorithm can make multiple traversals and aggregations between the same pair of up- and downstream jobs if multiple paths between two jobs exist. This operation should therefore be *associative*.
- *Downstream operation* (`ds_op`, optional): The downstream operation is the last step performed, after our iterative algorithm converges. For a job, it performs an aggregation on all of its downstream jobs and aggregated path weights.

Algorithm outline. We first preprocess the job dependency DAG with the AF `wt_fn` to generate the DAG edge weights `wt`. Then, for each job in parallel, our algorithm traverses the DAG and computes transitive closures along all paths, maintaining an "aggregated version" of `wt` using `e_op` and `p_op` along the way. Our algorithm completes in $O(log(diameter))$ iterations, where *diameter* is the longest path in the DAG. In each iteration, the algorithm maintains a *frontier* and a *base* table, both with the schema (`job, depOn, wt`). The frontier table records the set of discovered furthest reachable upstream jobs by `job` in `depOn`, while the base table records the set of all discovered reachable upstream jobs by `job` in `depOn`. The `wt` column of both tables records the aggregated weights along discovered paths from `job` to `depOn`. Each iteration joins and updates the frontier and base tables, extending the "reach" of each job by a maximum of 2×. Our algorithm pseudocode is shown in Algorithm 1.

4.5 Job value aggregation with Wing

4.5.1 Job value aggregation properties

Fair multi-hop time windowing. Aggregating value directly on even a single-hop time-windowed job dependency graph has a critical shortcoming: when considering multiple hops, jobs at the start of the observed trace still hold an advantage over jobs toward the end of the observation window in terms of opportunities to have their multi-hop downstream dependencies also land in the observation window. To better illustrate this, suppose we are given a recurring job template X with multiple jobs in our observation window. While ideally all jobs of X should have similar amounts of downstream dependencies, jobs of X that occur earlier in the trace are more likely to have their downstream dependencies also observed in the trace, while later jobs of X in the trace are more likely to have their downstream dependencies cut off due to the limits of using a static-length trace. In the limit of using an infinitely long trace, no time windowing is necessary.

A *multi-hop time window* is therefore needed to further

[9] Wing can fix priority inversions by raising the upstream job's priority before its dependent high-priority job arrives. Traditional OS methods require both jobs to have arrived at the scheduler, and dependency between the two jobs is communicated through concurrency data structures (e.g., locks). There is no lock-equivalent in Cosmos's scheduler.

```
// Helper functions
Function preprocess(s_hop) is
    gp_by_job = job G wts=wt_fn(depOn)(s_hop);
    return π job, depOn=wts.depOn, wt=wts.weights(gp_by_job);
end
Function extend_reach(t1, t2) is
    e_agg = π t1.job, t2.depOn, wt=e_op(t1.wt,t2.wt)(
                t1 ⋈ t1.depOn=t2.job t2);
    return job, depOn G p_op(wt)(e_agg);
end
// Computation start
Input : s_hop                          // Single-hop dependencies
i = 0;                                 // Iteration
ftr_i = preprocess(s_hop);             // Frontier
base = COPY(ftr_i) ;                   // Base
// base at the end of iter i covers deps up to 2^i hops
do
    i++;
    base_tmp = base − ftr_{i-1};
    ftr_i = extend_reach(ftr_{i-1}, ftr_{i-1});
    base_tmp = extend_reach(ftr_{i-1}, base_tmp) ∪ base;
    base = job, depOn G wt=p_op(wt) base_tmp;
    base = base ∪ ftr_i;
while COUNT(ftr_i) > 0;
return job, depOn G agg=p_op(wt) base;          // Converged
```

Algorithm 1: Multi-hop downstream analysis framework. `preprocess` first assigns weights to DAG edges with `wt_fn`. In each iteration, it calls `extend_reach` to further explore the graph from each job in parallel. In `extend_reach`, auxiliary edges with edge weights specified by `e_op` are created to denote newly discovered indirect dependencies (through the `JOIN`, or ⋈ operator). The auxiliary edges are deduplicated with a `GROUP BY` ($\mathcal{G}$) operator at the end of each iteration, yielding edge weights of `p_op(wt)`.

restrict the set of jobs eligible for value aggregation. Our multi-hop time windowing method works as follows: we first define a time window size ω smaller than the observation period. For each *valid* job j in the trace, we consider its entire set of directly and indirectly dependent jobs that are submitted by up to ω after its completion time. Here, we define valid jobs as jobs that complete at least ω prior to the end of the observation period. We set ω to *one week* for multi-hop dependency analysis, as the scale of the inter-job dependency graph bottlenecks transitive closure computation as ω increases: increasing ω exponentially increases the number of multi-hop inter-job dependencies to consider, as dependencies fan-out further into the future. ω is set to a week here to capture the majority of recurring dependencies that occur on a sub-weekly cadence (most recurring templates are submitted with inter-job arrival times of a day or less), while allowing our entire analyses pipeline to finish in approximately a day.

Value conservation. To conserve the total amount of value in the system, we employ an *equal contribution* scheme proposed in Owl [9], where each job contributes value to its directly-dependent upstream jobs equally, and the aggregate value of a job in this scheme is computed as the sum of value contributed upstream by all of its downstream jobs plus the value of the job itself. In this scheme, if a job j depends directly on the output of N jobs, it contributes $1/N$ of its value to each of its jobs directly upstream. Each of the N upstream jobs in turn further propagates j's (and their own) value upstream in the same fashion; e.g., if each of the N jobs directly depend on the output of M other jobs, j contributes $1/(N*M)$ of its value to each of the $N*M$ jobs two hops upstream. This yields the following equation, as proposed in Owl [9], for computing the aggregate value of a job:

$$agg_val(j) = \sum_{d \in \mathcal{D}_j} \left(\sum_{p \in \mathcal{P}_{(j,d)}} \prod_{e \in p} w_e * k_d \right) + k_j,$$

where $\mathcal{D}_j$ represents *all* downstream jobs of j, $\mathcal{P}_{(j,d)}$ represents all paths from j to d, w_e represents the weight of a directed edge e on the path p, and k_d and k_j represent the job-local values of d and j, respectively.

4.5.2 Wing value Aggregation Functions

We implement Owl's dependency-driven job valuation scheme with Wing's downstream multi-hop analysis framework, specifying Aggregate Functions as follows: the *weight function* `wt_fn` takes in a job j and its N upstream dependencies as input, and returns $1/N$ as the weight of each in-edge; the *edge operation* `e_op` multiplies the weights of its two operands; the *path operation* `p_op` sums the weights of its operands; and finally, for each job j, the *downstream operation* `ds_op` sums the job-local downloads of each job downstream of j multiplied by the aggregated path weights between the downstream job and j. j's job-local downloads are finally added to the downloads computed by `ds_op`, yielding j's aggregate downstream downloads.

Extensibility. While we elect to use downloads as a proxy for job value, Wing's framework is flexible enough to consider other metrics: e.g., if one day the dollar value associated with a job can be known, computing the aggregate downstream dollar value of a job is as easy as replacing a field in `ds_op`.

4.5.3 Aggregate value exploration and convergence

Using downloads as a proxy-metric for value, Fig. 6 shows the fraction of aggregate value explored in each iteration for each job on average. Considering the aggregate value of jobs with Wing allows us to uncover 83% of value that would otherwise be hidden if only job-local values (*iteration* = 0) were considered. In the context of value-based job scheduling (§5), this means that nearly 6× of value can be hidden from the scheduler if jobs are independently considered. The figure also shows that 99% of average job aggregate value can be explored within four iterations of our algorithm.

5 Wing-Agg: Inter-job value scheduling

Value scheduling. The objective in *value scheduling* is to maximize the value achieved from executing jobs in a workload, where the completion of each job is directly associated with an amount of *job-local value* attained. Job-local value can decay over time, and this behavior is often modeled as a *value function* (VF) in scheduling literature, which expresses value attained as a function of job completion time. In value

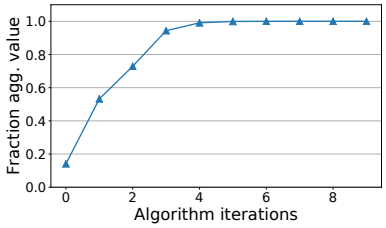


Figure 6: Aggregate value convergence. This figure shows the fraction of average aggregate job value uncovered downstream in each iteration of our value aggregation algorithm. 99% of aggregate value is discovered within four iterations.

scheduling, it is therefore important to complete jobs in a timely manner to achieve the most value.

Value and priority. We make a clear distinction between the terms *value* and *priority*. In this paper, we use the term *value* to describe a measure of "goodness" achieved associated with the completion of a job. *Priority*, on the other hand, defines the order in which pending jobs are assigned cluster resources: the higher the priority, the earlier a job receives its requested resources. Most commonly, including currently within Cosmos, the priority of a job is assigned by its submitter.

Wing-Agg. When inter-job dependencies are present, we find that it is important to consider the potential value downstream that can be lost if a job fails or is delayed. To consider the effects of inter-job dependencies, we propose a scheduling policy, *Wing-Agg*, that incorporates Wing's notion of inter-job dependencies into job priorities: *the goal of Wing-Agg is to achieve the most value for a given workload.*

As suggested in the introduction, completing the most value-impactful job may not lead to a scheduler attaining the most value, as some value-impactful jobs can also require large amounts cluster resource-time to complete. Indeed, prior work [8, 28, 44] has shown that schedulers can often benefit by considering together how much value a job provides and how much resource-time a job uses.[10]

Wing-Agg therefore considers the *aggregate value efficiency* of jobs, which measures how much aggregate value per aggregate resource-time a job impacts downstream. Essentially, Wing-Agg replaces user-assigned priorities with what Wing believes is a job's aggregate value efficiency. When a job arrives, Wing-Agg performs a look-up in the Wing-Store (§4.1). If the job is recurring, Wing-Agg computes the job's aggregate value efficiency by dividing the job's median historic aggregate value by its median historic aggregate compute-time, and assigns the quotient as the job's priority. If the job is ad-hoc, Wing-Agg estimates the job's aggregate value efficiency based on previous ad-hoc jobs that the same user has submitted. Wing-Agg assigns aggregate value efficiency rather than aggregate value as jobs' priorities to optimize for high value throughput.

[10] Although Wing-Agg and shortest-job-first both use job resource-time in their decisions, Wing-Agg frequently runs longer, more value-providing jobs ahead of shorter jobs.

6 Experimental setup

This section provides an overview of the Cosmos resource management infrastructure, describes our evaluated scheduling policies, and describes our experimental methodology.

Downloads attained as value. In our experiments, we use the number of downloads associated with the outputs of each job as a proxy for the value attained by a job. We model download attainment using real-world output download traces: if a job j completes at 1PM in the real-world (from the trace) but only completes at 2PM in our experiment, j attains only the output downloads associated with its outputs that occur after 2PM, and loses the downloads that occur between 1 and 2PM. A limitation of our model of value is that it does not reward completing a job early. Further research is required to determine how much additional value the early-completion of a job yields in data lakes.

Cosmos backend: YARN and hierarchical queues. Cosmos uses a *YARN-based resource manager* [12, 54] in the backend and utilizes *hierarchical queues* (queues, for brevity) to delineate resource boundaries between organizations—users/workflow managers can only submit SCOPE jobs to queues belonging to organizations of which they are a part. Cosmos uses a scheduling policy similar to the default policy that the *CapacityScheduler* in stock YARN uses, which orders jobs in each queue based on their (often user-) assigned priorities. A key difference is that jobs are scheduled with gang semantics in Cosmos—a job is admitted only when the scheduler can ensure that a user-provided minimum number of parallel, *job-requested* resources can be granted to it.

6.1 Simulation setup

We evaluate the application of Wing's analyses to scheduling using simulation-based experiments due to the scale of Cosmos: the Cosmos traces we use contain ~40k jobs per day, and ~160k inter-job dependencies. Experiments at this scale cannot realistically be attempted on research clusters without down-sampling jobs, at which point much inter-job dependency fidelity within the original workload will have been lost. We therefore use simulations to preserve the characteristics of inter-job dependencies in our experiments.

Simulation platform: design and implementation. Our simulation platform takes a discrete-event based approach. To ensure that our experiments retain most properties of YARN/Cosmos, our simulation platform makes minimal changes to the YARN architecture—our implementation only mocks out the real-time clock and the communication layers of the YARN servers. We also use real queue sizes for each hierarchical queue in our Cosmos cluster. The authors plan to contribute this simulator back to the open source community.

Simulation accuracy. To make simulation feasible given the scale of our job logs, the simulator does not model: (1) "internal" dependencies among stages of a job, but rather treat a job as a rigid collection of tasks; (2) resource-sharing through opportunistic execution [35] of job tasks, which allows jobs

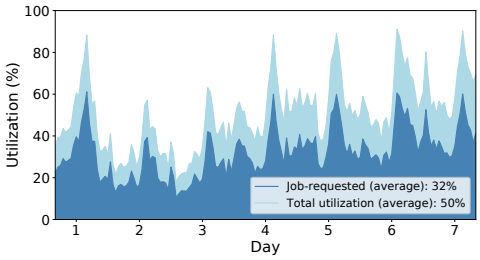


Figure 7: Cluster utilization. This figure shows the job-requested and total resource utilizations of our real cluster.

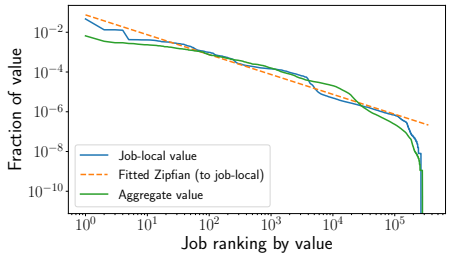


Figure 8: Distribution of job value. This figure shows the distributions of job-local value and aggregate job value, along with a Zipfian distribution fitted to job-local value. The distribution of job value deviates from Zipfian at lower job rankings.

to use more resources than requested when those resources are otherwise idle; and (3) job sizes based on resources used rather than job-requested resources, meaning that our simulations only consider the deep blue area in Fig. 7.

To evaluate the fidelity of our simulator, we measure the absolute differences in job completion times between jobs in our simulations (using the baseline system policies) and the same jobs run in the real cluster. We normalize the deltas by the job's real-world latency, and observe that even at the 99^{th} percentile, jobs are shifted by only 1.3% of their latencies. Our experiments run at 100% cluster capacity also achieve average resource utilization for job-requested resources within 1.5% of what is observed in the real cluster.

6.2 Evaluated scheduling policies

In addition to Wing-Agg (§5), we evaluate value-attainment on our workload traces on the following scheduling policies. All implement Cosmos's gang-scheduling semantics.

PRIO represents Cosmos's current approach, and is the default scheduling policy used by stock YARN in its CapacityScheduler. It orders jobs within each hierarchical queue based on user-specified priorities.

Wing-MIL. Millennium [8] is a VF-aware scheduler that orders jobs based on expected value attained per resource time: For each queued job it computes how much value can be gained at an estimated job completion time, divides the value by total job resource-time, and orders jobs by the resulting quotient. MIL is our implementation of Millennium on YARN, following descriptions in its design as closely as possible.

Wing-MIL is MIL using Wing-informed value functions (VFs): In addition to capturing how the job-local value of a single job decays, a Wing-informed VF captures *potential* value associated with the job lost over time by modeling a job's full decay of downloads. A job j attains all of j's aggregate downloads in the most optimistic case if it completes before or at its real-world completion time; otherwise, it loses value according to when users perform download operations *and* when downstream jobs fail due to it not completing on time (illustrated earlier in Fig. 5). For example, in a Wing-guided VF, if j completes at 1PM in the real-world but only completes at 2PM in our experiment, j loses all the direct downloads that occur between 1 and 2PM, *and* all the *indirect downloads* rooted in jobs that directly depend on j submitted between 1 and 2PM.

Plan-ahead based VF-aware policies. We attempted to evaluate more sophisticated plan-ahead based VF-aware policies, e.g., FirstOpportunityRate [44]. But, we found that one implementation of such a policy couldn't accommodate workloads at Cosmos scale, and efforts to mitigate bottlenecks by caching and limiting plan-ahead led to less value attainment than simpler policies (e.g., MIL). We therefore do not include our attempts with such a policy, as further work is warranted before conclusions are drawn.

6.3 Workload and predictor descriptions

Dataset. We use data from the final four weeks of our analysis dataset to evaluate our scheduling policies: Within the four weeks of data, Wing uses data in the first and second weeks to establish job and dependency profiles. Experiments are conducted over the third week, and downloads (value) are counted for each job up to one week (into the fourth week) from the completion of the job. Each day of traces contains ~40k jobs and ~160k inter-job dependencies.

Considering inter-job dependencies. Different from prior work, our experiments take characteristics of inter-job dependencies into account to realize more realistic workloads. For example, if a job holds a hard dependency on the output of an upstream job but the output is not available in time, the job fails due to missing input. Other dependency patterns, such as polling behavior (when a job waits for its inputs to become available), are also modeled faithfully. Jobs and dependencies considered in our experiments are described in Table 1.

Job value distribution. Job value, as measured by the number of downloads associated with the timely completion of a job in our experiments, are distributed roughly in a Zipfian fashion ($s = 1$) with deviation at the low end, as shown in Fig. 8. This means that the most valuable jobs are downloaded significantly more times than less valuable jobs. When scheduling for value on a workload that is inter-job dependency aware, schedulers should work to *unblock* the most valuable jobs before they arrive in order to attain their value.

Value efficiency predictor. Wing-Agg and Wing-MIL use a predictor to estimate the aggregate value efficiency associated with upcoming recurring jobs to optimize for value throughput. While §3 shows that direct inter-job dependencies can be predictable, it neither considers predictions on a

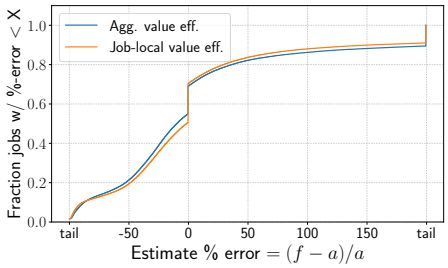


Figure 9: Value efficiency prediction. This figure shows the CDF of our predictor's performance on predicting the value efficiency and aggregate value efficiency of recurring jobs.

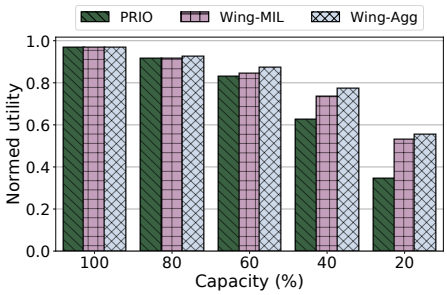


Figure 10: Benefits of Wing guidance. This figure shows the value attained for each scheduling policy, normalized to total value achievable. Wing-guidance (exemplified in Wing-Agg and Wing-MIL) is significantly beneficial at constrained capacities.

job's subgraph of downstream dependencies, nor a job's value impact. Evaluating predictions on aggregate value efficiency therefore allows us to better understand the performance of Wing-guided schedulers. For recurring jobs in our experiments, we use a median-based predictor to predict the value efficiency associated with a job. That is, given a recurring job j of template τ, we predict j's value efficiency based on the historical median value efficiency for jobs of template τ.

Fig. 9 shows the performance of our value efficiency predictor in a CDF. For predicting the aggregate value efficiency of a job, 39% of our predictions fall within ±20% of the actual value efficiency of a job, while for predicting the value efficiency of a single job, 44% of our predictions fall within ±20% of the actual value efficiency of a job. While we are working on further studies to improve predictor accuracy with more sophisticated methods, we find that the performance of our simple predictor enables Wing-Agg to outperform other evaluated scheduling policies in value attainment (§7).

7 Experimental results

We evaluate the efficacy of each scheduling policy for the actual full Cosmos resource capacity (100%) and for smaller capacities (at 80–20%). Value-attainment results are reported as a percentage of value achievable—i.e., if all jobs in workloads complete before any of their values are lost.

Cluster capacities & consequential policy decisions. Scheduling is most interesting when cluster capacity is constrained and schedulers need to make difficult decisions regarding which jobs to provide resources. Indeed, at 100% capacity, the baseline and more advanced schedulers perform similarly, completing > 99% of all jobs in the trace. We find that the lower cluster capacities (i.e., ≤ 40%) best exemplify the consequences of decisions a scheduler makes. We therefore focus the discussion of our results at these capacities to maximize observable differences.

Takeaways. Our experiments yield the following key takeaways. First, policies guided by Wing are better at achieving value when clusters are heavily-constrained. In particular, Wing-Agg outperforms all other compared policies at all capacities and improves value attained by up to 21% as capacity declines. Second, understanding the downstream impact of a job is crucial in constrained clusters, and that Wing-guided inter-job dependency predictions are accurate enough to be practical: Wing-Agg can effectively complete the prerequisites of the most consequential jobs. Finally, we demonstrate significant opportunity in applying inter-job dependency awareness in Wing to a cluster-wide queue and establishing a cluster-wide value metric: Wing-Agg achieves up to 93% of all value in our workload when using a single cluster-wide queue, using only 20% of cluster capacity.

7.1 Benefits of Wing guidance

Fig. 10 shows that policies guided by Wing beat PRIO at all capacities, with value attainment gaps widening as the cluster is increasingly stressed. At 60% capacity, Wing-Agg achieves 87% of value (vs PRIO's 80%). At 40% capacity, Wing-Agg achieves 77% of value (vs PRIO's 62%). Even at 20% capacity, Wing-Agg is able to capture more than half of all value (55%), while PRIO only captures 35% of value.

Considering aggregate value gives Wing-guided schedulers a two-fold benefit over PRIO. First, it naturally "fixes" priority mis-configurations, such as priority inversions, by propagating job value upstream, such that downstream jobs with high value are not blocked. Second, it guides schedulers toward sub-DAGs of high value efficiency jobs effectively, allowing schedulers to achieve more value with less resources.

Are ad-hoc jobs disadvantaged? Since Wing-Agg focuses on recurring jobs, we examine our logs to see if ad-hoc jobs are at a disadvantage when scheduled by Wing-Agg vs recurring jobs, where the priority of ad-hoc jobs are determined by the median aggregate value efficiency of previous jobs submitted by the same user. We find, from results at 20% cluster capacity, that 25% of recurring jobs fail, compared to 42% of ad-hoc jobs. However, recurring jobs also carry 9× more value than ad-hoc jobs. To optimize for value, Wing-Agg necessarily needs to complete larger fractions of recurring jobs. Indeed, recurring jobs are more often production jobs [34].

Dynamic priorities (Wing-MIL). Intuitively, policies using dynamic priorities (e.g., value functions, or VFs) such as Wing-MIL should perform better than static policies such as Wing-Agg, as VFs can express both importance and urgency while priorities only allow the expression of one of the two dimensions; but, we observe that Wing-Agg outperforms

Wing-MIL at all capacities, albeit only slightly.

Unlike Wing, which only depends on aggregate value-efficiency predictions, Wing-MIL also depends on the time-to-dependency predictions of directly-dependent jobs (§3) to determine when aggregate job value decays. But, while a part of this underperformance is indeed caused by imperfect predictions of time-to-dependencies, we find that providing Wing-MIL with perfect job value and time-to-dependency information does not help much. Further analyzing our results, we find that this underperformance is mainly due to Wing-MIL's failure to consider the *properties* of inter-job dependencies. For example, a downstream job that polls for the arrival of its inputs will not fail if its upstream jobs complete late. But, VFs constructed from historical data will still reflect a drop in value at the time the polling downstream job is expected to arrive, leading Wing-MIL to believe that it should give up prematurely on scheduling the job. This shortcoming can be addressed by considering dependency properties *explicitly*, but our attempted implementation of such a policy does not significantly improve over Wing-Agg: both Wing-Agg and our attempted implementation can complete the most impactful, value-efficient jobs in a timely manner.

Practicality of Wing-Agg. The simplicity of Wing-Agg is desirable from an engineering standpoint, as Wing-Agg is both highly practical and highly scalable: Integrating Wing-Agg into a production cluster requires minimal changes to the existing resource management framework, and all the information needed for Wing-Agg to determine a job's priority can be precomputed offline in Wing's analysis pipeline (§4). Adoption of Wing-Agg into production can therefore be straightforward, upon confirming job valuation schemes.

7.2 Sensitivity and ablation studies

Aggregate vs. job-local value. This section discusses benefits of understanding job value at an aggregate vs job-local level by comparing Wing-Agg against Wing-Direct, where *Wing-Direct considers the job-local value efficiency of a job*: i.e., Wing-Direct only considers direct-downloads associated with the outputs of and the compute-time of a single job only.

The patterned bars in Fig. 11 show the normalized value attained by Wing-Agg and Wing-Direct. While Wing-Direct outperforms PRIO, Wing-Agg maintains significant benefit over Wing-Direct at the tightest capacities: Wing-Agg attains 13% more overall value than Wing-Direct at 20% capacity. Our analysis finds that Wing-Direct's knowledge of job resource consumption allows it to effectively complete jobs at the head of queue, enabling it to complete a similar amount of jobs as Wing-Agg. But, with knowledge of historical aggregate value efficiency, we find that Wing-Agg completes jobs in the more value-heavy sub-DAGs of the inter-job dependency DAG, yielding significant improvements over Wing-Direct.

Wing predictions vs. Oracle knowledge. We examine how much potential benefit better predictions can provide to each Wing-aided policy. *Oracle Wing-Agg* represents Wing-Agg endowed with perfect knowledge of aggregate value efficiency, and *Oracle Wing-Direct* represents Wing-Direct provided with perfect knowledge of job-local value efficiency.

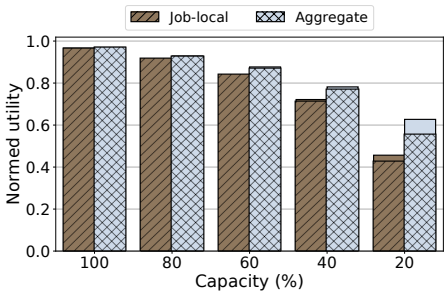


Figure 11: Benefits of aggregate job value. *Aggregate* (corresponding to aggregate download-aware) vs *Job-local* (corresponding to direct download aware only) bars show the benefits of aggregate value, compared to only scheduling based on job-local value. The solid portion of the bars show the benefits of Oracle knowledge.

While we find that having better predictions are beneficial, the differences between the solid (representing policies with Oracle knowledge) and the patterned bars (representing policies with Wing-provided predictions) in Fig. 10 and Fig. 11 show that at most capacities, Wing-guided schedulers achieve close to the value attained by their Oracle variants. However, having more accurate information presents opportunity for significant gain in value attained for Wing-Agg at 20% capacity: e.g., Oracle Wing-Agg improves value realized over Wing-Agg by 8% of overall value. Conversely, although Oracle Wing-Direct is granted exact knowledge of how value-efficient each job is, its view of the overall inter-job dependency graph leads to only incremental benefits.

Oracle benefits to aggregate value aware policies come from a more accurate knowledge of a summarized view of the inter-job dependency graph: compared to single job value-aware policies with Oracle knowledge, a policy such as Oracle Wing-Agg can efficiently complete the most consequential jobs in the job dependency graph, increasing value attained (by up to 18% of overall value vs Oracle Wing-Direct) and reducing the number of jobs failed due to missing input (by 3% of all jobs vs Oracle Wing-Direct).

Sensitivity to mis-predictions. We examine the sensitivity of Wing-Agg to aggregate value efficiency mis-predictions on our workload by running experiments that introduce artificial shifts in aggregate value efficiency provided by Oracle Wing, using 20% cluster capacity. Each experimental run is associated with a maximum artificial shift s, where $s \in \{1.1, 1.25, 1.5, 2, 5, 10\}$. For each job j within each run, we scale the aggregate value efficiency e_{val} of j provided by Oracle Wing by multiplying e_{val} by a randomly sampled multiplier m between $1/s$ and s. Our results show that Wing-Agg is not sensitive to mis-predictions in value on our workload: For $s \leq 5$, value attained is only reduced by at most 4% vs Oracle Wing-Agg. For $s = 10$, value attained is only reduced by 11%. This insensitivity is because job values in our workload are distributed in a Zipfian fashion (§6.3), where the most

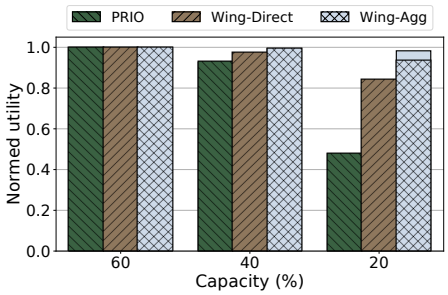


Figure 12: Benefits of Wing-guidance with a cluster-wide queue. This figure shows the value attained for policies from 60–20% cluster capacities in a cluster with a merged cluster-wide queue. All policies complete all jobs at 60% capacity. Wing-guidance (exemplified by Wing-Agg) is increasingly beneficial at lower capacities. The solid portion of the bars show the benefits of Oracle knowledge.

valuable jobs are much more valuable than other jobs.

Reducing transitive closure computation. At 20% capacity, Wing-Direct (0 iterations of Wing's multi-hop analysis) attains 42% of all value, while Wing-Agg (9 iterations executed) attains 55% of all value. In Fig. 6 in §4.5.3, we find that 99% of aggregate value of most jobs can be explored in four iterations of Wing's multi-hop analysis. We therefore believe that four iterations of exploration would be sufficient to similarly attain 55% of all value, and that two iterations of exploration would allow us to attain close to 50% of all value.

7.3 Cluster-wide queue and value metrics

Our earlier results correspond to a simplified view of Cosmos using strictly enforced queue boundaries. Hard queue boundaries restrict placement more than in the real system, where resource-sharing (§6.1) softens queue boundaries, which might exaggerate Wing-Agg's benefits. To confirm that Wing-Agg's improvements are not due to hard queue boundaries, we evaluate a boundary-free alternative with experiments run using a single global, cluster-wide queue.

Evaluation. Fig. 12 shows the value attainment of our evaluated scheduling policies using a single cluster-wide-queue. We note that all jobs are able to complete for all scheduling policies at 60% cluster capacity. Indeed, the dark blue area in Fig. 7 show that these requests peak at around 60%. At 40% capacity, the cluster still has more capacity than needed most of the time: Wing-Agg achieves 99% of value, and Wing-Direct and PRIO achieve 97 and 93% of value, respectively.

Under extreme capacity crunch (e.g., 20% capacity), removing restrictions of hard queue boundaries improves value attained of all policies. But, a Wing-guided scheduler sees significantly more benefit in terms of absolute value achieved. With a cluster-wide queue at 20% capacity, Wing-Agg attains 93% of value, whereas Wing-Direct attains 84%, and PRIO only attains 47%. Furthermore, Wing-Agg fails fewer jobs compared to both Wing-Direct and PRIO (11% vs 13% and 25% of jobs, respectively).

We find that understanding inter-job dependencies is critical, as Wing-Direct with Oracle knowledge did not significantly outperform Wing-Direct with predicted values, both in terms of value attained and in terms of number of jobs failed; yet, we find that Wing-Agg with Oracle knowledge, in this setting, can achieve up to 98% of all value (comparable to performances at 100% capacity), while failing only 7% of all jobs (compared to 26% in a multi-queued setting at 20% capacity). One of the reasons why Wing-Agg is able to attain 93% of all value using only 20% of cluster capacity is due to its ability "unblock" the most valuable downstream jobs.

Recall that the simulated job sizes in our experiments are based on job-requested resources, rather than job-used resources, which may be higher because of opportunistic execution. As a result, cluster utilization is lower in our experiments. But, we believe that the rankings of the different schedulers are not affected, because the number of opportunistic resources highly correlate with that of allocated job-requested resources, both across the top 10% of most valuable jobs (Spearman correlation of 0.85) and across all jobs (Spearman correlation of 0.84). Indeed, the amount of opportunistic resources available to a job is capped with a max proportional to the number of allocated job-requested resources [49]. So, the relative differences shown for 20% cluster capacity may instead be for 30% cluster capacity in the heavier workload.

Toward establishing a cluster-wide value metric. Our results confirm that removing queue boundaries would be beneficial. Partitioning resources into queues naturally introduces resource fragmentation, but usage of queues is often viewed as a "necessary evil," as certain organizations are willing to pay more to have guaranteed access to their share of compute. Yet, näively removing queue boundaries without a quota-system [55] in place may introduce resource competition, where users across different organizations assign increasingly high priorities to their jobs to acquire guaranteed resources. A cluster-wide, automated arbitrator that understands both system-internal (e.g., aware of downstream number of affected jobs and user-downloads) *and* organizational/user-defined notions of importance is therefore required. We see this as an exciting direction for further research.

Current state of deployment. Instead of immediately deploying Wing-Agg as described, the Microsoft Cosmos resource management team has asked us first to deploy an inter-job dependency advisory tool using analyses from Wing, to aid users on better configuring their jobs. The tool will allow us to gather user feedback on our recommendations.

8 Related work

Workflow managers. Workflow management for batch analytics jobs is a widely studied area in the fields of databases and data management [30, 40, 41]. Our work differs in two primary ways: (1) workflow managers often assume the availability of a dependency graph up-front, while Wing infers properties of inter-job dependencies from job history; and (2) workflow managers optimize only a single pipeline of jobs submitted by one user at a time, while Wing considers inter-

dependent jobs across workflow and organization boundaries.

Cluster workload analysis. Although much work has been done on cluster workload analysis from many different perspectives (e.g., resource/workload heterogeneity [1, 11, 26, 37, 45], failure analysis [7, 17, 46], job predictability [43, 50, 52], and intra-job task dependency [23, 24, 51]), most prior work assumes (implicitly or explicitly) that each job is independent of other jobs. This paper fills the knowledge gap with analyses of inter-job dependencies and application of this knowledge in cluster scheduling.

Cluster scheduling. Although a variety of work has been published in the area of cluster scheduling, each trying to address scheduling woes of different kinds of workloads (e.g., support for general batch analytics [5, 10, 18, 21, 22, 29, 34, 43, 53, 54], low latency scheduling [15, 16, 35, 42], and strategies to handle mixes of workloads [12, 19, 20, 48, 55]), most work in cluster scheduling similarly assume the independence of jobs. Our work shows that incorporating knowledge of inter-job dependencies can improve cluster scheduling in an environment with a lot of data and work product sharing, and we believe that considering inter-job dependencies can help future schedulers better tackle challenges, such as enabling better job task placement and learning better scheduling policies [38, 47].

Task-DAG schedulers assign resources to inter-dependent tasks within a job based on knowledge of the overall task-DAG [18, 23, 24, 38]. Such techniques and our proposed policies can be complementary, as task-DAG schedulers drill into job-level details while our schedulers (e.g., Wing-Agg) work at a higher level and treat jobs as black boxes. In particular, schedulers that predict the arrival of future jobs [34, 38] can benefit from the availability of inter-job dependency context to refine their predictions. Some task-DAG scheduling techniques could also be applied to the problem of inter-job dependency scheduling; but, these task-DAG schedulers generally assume upfront availability of task-DAGs, while full inter-job dependency graphs are rarely available ahead of time. An interesting direction for future research is in combining task-DAG scheduling techniques with some form of Wing-provided "probabilistic inter-job dependency" DAGs.

Jockey and Morpheus. Jockey [18] uses the direct dependencies of jobs to illustrate the importance of maintaining low job latency variance, but uses a step-function with value=1 until the *user-provided deadline* as each job's value function (VF). Morpheus [34] improves upon Jockey's notion of VFs by deriving deadlines based on a job's first consumer (as observed from historical instances of that job), but still considers all jobs as equal in value. In addition to our characterization of inter-job dependencies in a large analytics cluster, our work extends Morpheus and Jockey in two ways: (1) jobs no longer all have the same value—instead, Wing derives each job's value (and therefrom priority) as the sum of a chosen value metric (e.g., downloads) for all downstream dependencies, and (2) value is no longer a step-function with a single deadline based on a job's first direct consumer, but a rich decay proportional to the aggregate value of dependency sub-DAGs rooted in each direct consumer. While we do not directly compare against Morpheus, in §7.1, we find, in the context of Wing-MIL, that a premature drop in aggregate value can lead to the scheduler giving up early when dependency properties are not considered, leading to lower value attainment. Considering value as a step-function with a single deadline can therefore potentially be detrimental when inter-job dependencies are present in cluster workloads. While Wing-Agg uses only the initial "height" of the aggregate value VF of each job to set priorities, we believe that full aggregate value VFs can still better guide other scheduling decisions, such as determining which jobs to load-shift.

Systems using job recurrence and data provenance. There has also been much prior work on systems that efficiently collect provenance data [13, 39] and systems that both exploit job recurrence and data provenance on other problems [25, 33], such as garbage-collecting shared computation results. Our work uses similar ideas, but focuses on facilitating better value attainment in resource scheduling.

Owl and Guider. Our previous work Owl [9] and Guider [39] introduced the usage of job dependencies to determine the value of jobs. Wing operationalizes and expands upon prior work by (1) analyzing and characterizing inter-job dependencies in a large cluster, (2) evaluating predictability of recurring inter-job dependencies, (3) integrating inter-job dependencies into cluster schedulers, (4) applying said schedulers to a real scheduling problem, and (5) providing a general aggregate inter-job dependency analysis framework.

9 Conclusion

Complex inter-job dependencies pervade modern data lakes, creating complex problems as cluster schedulers make decisions without knowing of them. The Wing dependency profiler uncovers these dependencies from provenance logs and provides improved guidance to cluster schedulers. Evaluations with real job traces show that significantly more value, in terms of successful user downloads, can be attained by using Wing-guided priority assignments over those provided by users. Wing's effectiveness opens a new range of resource management possibilities guided by automatically-determined knowledge of the impact of jobs.

Acknowledgements

We thank John Wilkes (our shepherd) and our OSDI 2020 reviewers for their valuable feedback and suggestions. We also thank Raghu Ramakrishnan, Boris Asipov, Hiren Patel, Yiwen Zhu, Isha Tarte, and Panagiotis Garefalakis for their help throughout the development of this project. We thank the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure, Salesforce, Samsung, Seagate, Two Sigma, and Western Digital) and VMware for their interest, insights, feedback, and support.

References

[1] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18. USENIX Association, 2018.

[2] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10. ACM, 2010.

[3] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation : Practice and Experience*, 24(13), September 2012.

[4] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, CIDR '15, January 2015.

[5] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[6] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2), August 2008.

[7] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*, ISSRE '14. IEEE Computer Society, Nov 2014.

[8] Brent N. Chun and David E. Culler. User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02. IEEE Computer Society, May 2002.

[9] Andrew Chung, Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Panagiotis Garefalakis, and Gregory R. Ganger. Peering Through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19. ACM, 2019.

[10] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware Container Scheduling in the Public Cloud. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[12] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19. USENIX Association, February 2019.

[13] Sergio Manuel Serra da Cruz, Patricia M. Barros, Paulo Mascarello Bisch, Maria Luiza Machado Campos, and Marta Mattoso. Provenance Services for Distributed Workflows. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08. IEEE Computer Society, 2008.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, OSDI '04. USENIX Association, 2004.

[15] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[16] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15. USENIX Association, 2015.

[17] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *Proceedings of*

the IEEE 37th International Conference on Distributed Computing Systems, ICDCS '17. IEEE Computer Society, June 2017.

[18] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12. ACM, 2012.

[19] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[20] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18. ACM, 2018.

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11. USENIX Association, 2011.

[22] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[23] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[24] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016.

[25] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI '10, 2010.

[26] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the 2019 International Symposium on Quality of Service*, IWQoS '19. ACM, 2019.

[27] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16. ACM, 2016.

[28] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04. IEEE Computer Society, June 2004.

[29] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. ACM, 2009.

[30] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, March 2011.

[31] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting Subexpressions to Materialize at Datacenter Scale. *Proceedings of the VLDB Endowment*, 11(7):800–812, March 2018.

[32] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[33] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. ACM, 2018.

[34] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, November 2016.

[35] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15. USENIX Association, 2015.

[36] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09. IEEE Computer Society, Sep. 2009.

[37] Qixiao Liu and Zhibin Yu. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[38] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the 2019 ACM Special Interest Group on Data Communication*, SIGCOMM '19. ACM, 2019.

[39] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Phil Cudre-Mauroux. Dependency-Driven Analytics: a Compass for Uncharted Data Oceans. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, January 2017.

[40] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10. ACM, 2010.

[41] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proceedings of the IEEE 26th International Conference on Data Engineering*, ICDE '10. IEEE Computer Society, March 2010.

[42] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. ACM, 2013.

[43] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18. ACM, 2018.

[44] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05. IEEE Computer Society, 2005.

[45] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. ACM, 2012.

[46] Andrea Rosà, Lydia Y. Chen, and Walter Binder. Predicting and mitigating jobs failures in big data clusters. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '15. IEEE Computer Society, 2015.

[47] Malte Schwarzkopf and Peter Bailis. Research for Practice: Cluster Scheduling for Datacenters. *Communications of the ACM*, 61(5):50–53, April 2018.

[48] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. ACM, 2013.

[49] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. Autotoken: Predicting peak parallelism for big data analytics at microsoft. *Proceedings of the VLDB Endowment*, 13(12):3326–3339, August 2020.

[50] Liqun Shao, Yiwen Zhu, Siqi Liu, Abhiram Eswaran, Kristin Lieber, Janhavi Mahajan, Minsoo Thigpen, Sudhir Darbha, Subru Krishnan, Soundar Srinivasan, and et al. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-Based Platforms. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[51] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[52] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A Kozuch, and Gregory R Ganger. JamaisVu: Robust scheduling with auto-estimated job runtimes. Technical report, Technical Report CMU-PDL-16-104. Carnegie Mellon University, 2016.

[53] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16. ACM, 2016.

[54] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13. ACM, 2013.

[55] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, EuroSys '15. ACM, 2015.

[56] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, San Jose, CA, 2012. USENIX Association.