Peering through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters

Andrew Chung
Carnegie Mellon University

Carlo Curino
Microsoft

Subru Krishnan
Microsoft

Konstantinos Karanasos
Microsoft

Panagiotis Garefalakis
Imperial College London

Gregory R. Ganger
Carnegie Mellon University

Abstract

Shared multi-tenant infrastructures have enabled companies to consolidate workloads and data, increasing data-sharing and cross-organizational re-use of job outputs. This same resource- and work-sharing has also increased the risk of missed deadlines and diverging priorities as recurring jobs and workflows developed by different teams evolve independently. To prevent incidental business disruptions, identifying and managing job dependencies with clarity becomes increasingly important. Owl is a cluster log analysis and visualization tool that (i) extracts and visualizes job dependencies derived from historical job telemetry and data provenance data sets, and (ii) introduces a novel job valuation algorithm estimating the impact of a job on dependent users and jobs. This demonstration showcases Owl's features that can help users identify *critical job dependencies* and quantify *job importance* based on jobs' impact.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering →** *Scheduling*.

Keywords

cloud computing; cluster scheduling

ACM Reference Format:
Andrew Chung, Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Panagiotis Garefalakis, and Gregory R. Ganger. 2019. Peering through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3299869.3320239

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5643-5/19/06...$15.00
https://doi.org/10.1145/3299869.3320239

1 Introduction

Large data-driven companies such as Microsoft operate complex, unified *data lakes* to promote data sharing and to minimize data-access barriers. At Microsoft, the internal Cosmos infrastructure consists of large clusters with tens of thousands of nodes each, running more than 500 thousand jobs daily, many of which consume data across organizational boundaries [4]. In such shared infrastructures, the vast majority of the data are processed by *recurring workflows*, each of which consists of a set of one or more inter-dependent jobs typically encapsulating a logical business task. The loose boundaries of data lakes, and the widely shared nature of data within, hastens the need for more principled management of job and data dependencies.

In companies with many users and sizable shared data repositories, job owners are seldom aware of who consumes their jobs' outputs, as their knowledge of job dependencies is often limited by their scope of business interactions and contracts. At the same time, seemingly harmless modifications to *upstream* jobs (e.g., submission schedule, priority, and output schema) can produce a ripple-effect on *downstream* jobs, interrupting chains of inter-dependent jobs and workflows and causing businesses significant amounts of damage. Such obscure job dependencies often require co-dependent teams to establish carefully negotiated service level agreements, a counter-productive process towards the goal of a universal data lake.

This demo showcases Owl, an end-to-end system that extracts, analyzes, and visualizes job dependencies from hundreds of terabytes of job telemetry and data provenance logs generated by Microsoft's Cosmos clusters daily. We will walk attendees through Owl's job analysis workflow (§2) and show how Owl can help solve the *obscured dependency problem*:

(1) Owl exposes job dependencies. Analyzing provenance data, Owl reveals job dependencies within-and-across workflows (§3.2) that are otherwise hidden in popular workflow visualizers [1, 2]. Owl also visualizes recurring job dependencies (§3.3), enabling users to explore dependency metrics (e.g., time-to-dependency, number of priority inversions) between frequently dependent jobs and use these metrics to fine-tune job hyper-parameters (e.g., priority, resources allocated).

(2) Owl evaluates jobs based on job impact. The potential of jobs to disrupt subsequent processes led us to develop a concept of *job importance* based on a job's historical downstream dependencies. Using a novel algorithm, Owl quantifies a job's potential downstream impact (§3.4) in two dimensions: (1) *work impact* measures a job's effect on other jobs and (2) *user impact* measures its effect on cluster users. While other systems have tackled the woes of managing the sharing of large amounts of data from the perspectives of data curation, discovery, and provenance [3, 5, 6, 8], to the best of our knowledge, Owl is the first to perform job valuation by combining job telemetry with data provenance.

With a clear view of job dependencies and quantified measures of job downstream impact, users can optimize their jobs with a better understanding of the consequences of their modifications, while avoiding the disruption of "valuable" jobs. We conclude our demo by briefly discussing how dependency-driven job importance can be used to make job scheduling more efficient (§4).

2 Owl workflow

This section provides a brief overview of Owl's Cosmos job data analysis workflow. To extract job data, Owl relies on two external components: (i) *ProvMap* provides single-hop file-level provenance on Cosmos jobs in the form of the sets of files consumed and produced by a job, and (ii) *JobRepository* provides job-level telemetry such as run time, CPU-hours, and data read/written.

Owl ingests file provenance from ProvMap and generates single-hop job-to-job dependency relations using the last-writer-wins policy[1]. Then, it analyzes logs from JobRepository to produce high-level job metadata (e.g., job recurrence statistics and workflow specifications). Using both single-hop job dependencies and high-level job metadata, Owl infers both *workflow* (§3.2) and *recurring dependencies* (§3.3).

Single-hop job dependencies are also used to iteratively construct the job dependency graph. The graph, along with processed job metadata, is used to evaluate job importance based on the downstream impact (§3.4).

Finally, output generated is exported to a SQL database for consumption by the front end. Intermediate output of all steps above are written to Cosmos's distributed file system.

3 Owl features

This section describes features that demo attendees will be able to experience when interacting with Owl.

3.1 Background: representing dependency

Job dependencies can be represented as a *weighted directed acyclic graph (WDAG)*, where each *vertex* is an instance of a job and each *edge* represents data-flow between a pair of job instances (where data is generated by the source and consumed by the target vertex). The *weight* of an edge represents the reliance of a target job on a source job, and is determined by a flexible *weighting scheme*. We use the `equal` weighting scheme for our demo, where each source job is viewed as of equal importance to a downstream target job. In other words, if a job depends on N other jobs, it would have N incoming edges, each weighted $1/N$.

3.2 Workflow dependencies

Motivation. Users often submit *workflows*, or sets of inter-dependent jobs, to complete complex business tasks. Although these business tasks might similarly be achievable with a single monolithic job, breaking the job into smaller component jobs allows for improved code reusability, manageability, and debuggability. These workflows are usually managed by automated time-based job scheduling systems.

In a large organization like Microsoft, it is virtually impossible for users to recognize all consumers of a job's outputs. Debugging workflows can therefore be a daunting task, as workflows can include numerous job dependencies across multiple workflows, each owned by a different team. The lack of awareness of dependent jobs external to a workflow can lead to job disruptions. For example, since a workflow owner often only cares about the end-result of their workflow (i.e., when their business task completes), intermediate jobs can be altered without warning as long as the pipeline completes end-to-end. These seemingly harmless modifications can potentially cause external job failures unbeknownst to the workflow owner due to inconspicuous inter-workflow dependencies. Owl's workflow view clarifies dependencies within-and-across workflows.

Visual features. Owl's *workflow graph view* allows users to browse important properties of their workflows using an interactive graph (Fig. 1). Graph vertices (workflow jobs) can be resized proportionally to a selected job attribute to help users identify outliers or anomalies. For instance, a user can size vertices with respect to job runtime to identify execution bottlenecks or with respect to CPU-time utilized to analyze resource budget consumption. A standard *Gantt chart* shows users when each workflow job is submitted and how long it runs for during the workflow's lifetime.

The *inter-workflow dependency graph* allows users to discover dependencies across workflows. With the graph, users can easily identify owners and properties of upstream and downstream workflows, as well as pinpoint problematic cross-workflow job dependencies.

The *workflow diff utility* enables side-by-side comparison between two executions of the same recurring workflow using interactive graphs. It allows users to effectively identify anomalies in workflow structures, such as when a workflow instance executes more jobs than usual.

[1] Job A depends on B if A takes f, an output of B, as input. If both B and C produce f, A depends on the job that last wrote to f before its start time.

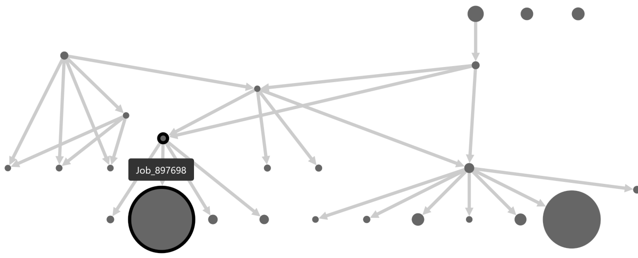

Figure 1: Workflow graph Shows the job dependency structure within a workflow. Allows users to identify important jobs by auto-sizing job vertices with respect to job execution attributes such as CPU-time (depicted).

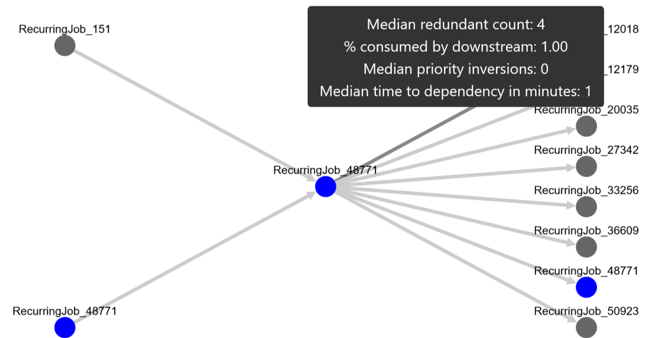

Figure 2: Recurring Job dependency graph Displays the target recurring job (center) and its upstream (left) and downstream (right) recurring jobs. Hovering over an upstream/downstream link shows statistics of the dependency.

3.3 Recurring job dependencies

Motivation. *Recurring jobs*, or jobs that are repeatedly submitted over time to analyze fresh data, make up the majority (~60%) of CPU-time utilized in Microsoft's Cosmos clusters [7]. When recurring jobs depend on each other, a *recurring dependency* is introduced and an implicit contract is formed between the pair of jobs. Breaking the contract in any way can potentially lead to service disruption downstream. It is thus in a job owner's interest to understand characteristics of upstream recurring jobs and be aware of recurring jobs consuming the job's output.

Visual features. Aside from showing basic recurring job attributes (e.g., periodicity and distribution of job execution properties), Owl's *recurring jobs view* features a *recurring dependency graph* (Fig. 2) that allows users to navigate and analyze recurring dependencies and their statistics (also displayed in an interactive distribution chart), both upstream and downstream.

With the dependency graph and statistics, users will be able to discern important dependencies. For example, using the *time-to-dependency* statistic, users will be able to see which upstream job their recurring job is frequently waiting on and get a sense of their job's *deadline slack* with respect to the submission time of downstream jobs. Using the *percent-consumed* statistic, determined by the number of job instances consumed by a recurring downstream job over the total number of job instances completed, users can get a sense of which jobs upstream are critical to their job as well as which jobs downstream their job is critical to.

3.4 Job impact

Motivation. Modifying job properties (e.g., output schema) without considering job dependencies can cause a ripple effect downstream, leading to job delays or even failures. As companies move towards universal data lakes that promote heavy data sharing, altering jobs without breaking downstream dependencies becomes a difficult task. While avoiding job disruption is always preferred, in a production setting it is critical, as job failures can affect business continuity.

To provide users a sense of how important a job is, in terms of its potential effect on downstream operations, we introduce the *job impact score*, derived from historical job dependencies and telemetry logs. The job impact score allows users to quickly discern how many downstream operations their job affects, as well as *which* downstream operation, if affected, will lead to the most potential operation disruption further downstream.

There are two main *facets* to our job impact score: (1) the *work impact* facet measures the impact of a job on downstream jobs in terms of historical CPU-time blocked (i.e., computation-hours that cannot proceed due to the upstream job not completing by the submission time of the downstream job). (2) The *user impact* facet measures the impact of a job on users in terms of the historical number of downloads blocked (i.e., file views that cannot happen due to the upstream job failing to produce its output(s)). The remainder of the subsection briefly walks the reader through our scoring methodology and how Owl visualizes historical job impact.

Methodology: evaluating job impact. Owl presents a novel method for quantifying different facets of job importance fairly using jobs' historical downstream dependencies. Each job j starts out with a *base score* k_j. The base score corresponds to a job run statistic — for example, the statistic used to compute *work impact* is CPU-time while the statistic used to compute *user impact* is number of downloads. Other statistics, such as bytes read/written, can easily be incorporated.

Downstream jobs *contribute* scores upstream, where the amount of contribution a downstream job makes to an upstream job is determined by the edges on the path(s) between the jobs and by the base score of the downstream job.

The impact score[2] of a job j can be summarized as follows:

$$score(j) = \sum_{d \in \mathcal{D}_j} min(1, \sum_{p \in \mathcal{P}_{(j,d)}} \prod_{e \in p} w_e) * k_d + k_j, \quad (1)$$

[2]Only deduping and summing the base scores of downstream jobs [8] is inflexible and disproportionately promotes jobs with high degrees of fan-in.

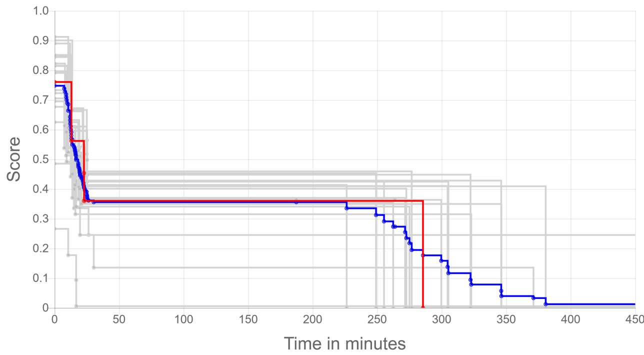

Figure 3: Job utility function graph Shows the value (score) of a job as a function of time-from-submission. The score displayed is normalized to the score of the most valuable job in the hierarchical queue. The red line displays the utility function of the user-selected job, while the gray lines represent utility functions of other instances of the same recurring job. The blue line sketches the average score over time of the recurring job.

where $\mathcal{D}_j$ represents *all* downstream jobs of j (single-hop or otherwise), $\mathcal{P}_{(j,d)}$ 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 base scores of d and j, respectively.

Methodology: discovering dependencies. Computing the job impact score (Eq. 1) requires discovering the transitive closure of each job in the job dependency WDAG while maintaining edge weights along the way. We use an iterative bulk-synchronous-parallel algorithm to compute the transitive closures.

Our algorithm starts out with single-hop job dependencies provided by ProvMap. In each iteration, each vertex maintains a set of *frontier* vertices, denoting its set of furthest "reachable" vertices in the iteration, and a set of *base* vertices, corresponding to its set of reachable vertices that are not in the frontier. Each vertex then expands its reach by 2x hops by querying its frontier vertices for *all* of their reachable vertices and updating its frontier and base sets correspondingly. With the algorithm, a job would start out knowing its dependencies directly downstream. It would then discover dependencies two hops downstream in the first iteration, four hops downstream in the second iteration, and so on.

The algorithm proceeds until no new vertices are discovered in the frontier of any vertex, and can be shown to converge in $O(log(diameter))$ iterations. We deploy our algorithm in Cosmos and run it with 2k parallel tasks, converging in 10 iterations over a month of cluster data.

Visual features. The job view, aside from providing a summary of job statistics, features the job impact view.

Job impact is visualized in two graphs. (1) The *job utility function graph* (Fig. 3) allows users to get a better sense of the urgency of their jobs. Each drop in score on the red/gray lines in the figure corresponds to the submission of a downstream dependent job relative to the submission time of the selected job. The magnitude of each drop corresponds to the value of the submitted downstream job. Naturally, users would want their jobs to complete before a downstream dependent job with high value is submitted, hence enabling users to infer a "deadline" for their jobs. (2) The *Sankey graph* (Fig. 4) allows users to quickly identify important downstream dependencies. The graph shows how value flows through job dependencies, where the height of a vertex represents the importance of a job, while the width of a flow between two vertices measures how much value a downstream job contributes upstream. For both graphs, users can select between our two facets of job impact: *work impact*, which is measured in CPU-hours, and *user impact*, which is measured in output downloads by users.

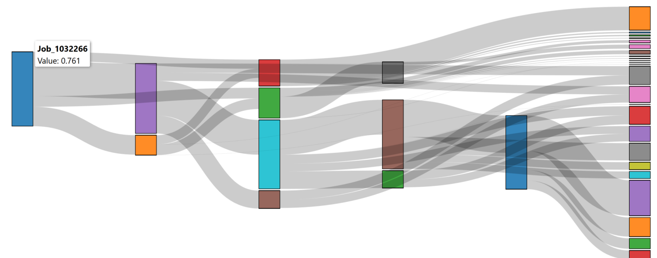

Figure 4: Interactive Sankey graph Shows how downstream jobs contribute value upstream. Each vertex is a job, with the height of the vertex representing relative job value. Hovering over a job displays its name and value The root job (left) represents the user-selected job. Clicking on leaf jobs (right) expands the graph further downstream.

4 Future work

Today, many systems schedule jobs based on user-provided priority metrics without consideration of inter-job dependencies and how to most effectively complete jobs that users care the most about. Squinting a bit, we can see how the Owl front end allows users to make these prioritizations manually, via adjustments to job hyper-parameters. An intuitive direction for our future work is therefore to improve our current dependency-based job valuation scheme and to incorporate it in real job scheduling systems — driving these systems toward an automated, data-driven prioritization scheme.

References

[1] 2019. Apache Airflow. https://airflow.apache.org/.
[2] 2019. Luigi. https://github.com/spotify/luigi.
[3] A. Bhardwaj et al. 2015. Datahub: Collaborative data science & dataset version management at scale. In *CIDR*.
[4] C. Curino et al. 2019. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*.
[5] A. Halevy et al. 2016. Goods: Organizing Google's datasets. In *SIGMOD*.
[6] J. M. Hellerstein et al. 2017. Ground: A Data Context Service. In *CIDR*.
[7] S. A. Jyothi et al. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI*.
[8] R. Mavlyutov et al. 2017. Dependency-Driven Analytics: A Compass for Uncharted Data Oceans. In *CIDR*.