Data Jockey: Automatic Data Management for HPC Multi-Tiered Storage Systems

Woong Shin*, Christopher D. Brumgard*, Bing Xie*, Sudharshan S. Vazhkudai*,
Devarshi Ghoshal[†], Sarp Oral*, Lavanya Ramakrishnan[†],
*Oak Ridge National Laboratory
Email: {shinw, brumgardcd, xieb, vazhkudaiss, oralhs}@ornl.gov

[†]Lawrence Berkeley National Laboratory
Email: {dghoshal, lramakrishnan}@lbl.gov

***Abstract*—We present the design and implementation of Data Jockey, a data management system for HPC multi-tiered storage systems. As a centralized data management control plane, Data Jockey automates bulk data movement and placement for scientific workflows and integrates into existing HPC storage infrastructures. Data Jockey simplifies data management by eliminating human effort in programming complex data movements, laying datasets across multiple storage tiers when supporting complex workflows, which in turn increases the usability of multi-tiered storage systems emerging in modern HPC data centers.**

Specifically, Data Jockey presents a new data management scheme called "goal driven data management" that can automatically infer low-level bulk data movement plans from declarative high-level goal statements that come from the lifetime of iterative runs of scientific workflows. While doing so, Data Jockey aims to minimize data wait times by taking responsibility for datasets that are unused or to be used, and aggressively utilizing the capacity of the upper, higher performant storage tiers.

We evaluated a prototype implementation of Data Jockey under a synthetic workload based on a year's worth of Oak Ridge Leadership Computing Facility's (OLCF) operational logs. Our evaluations suggest that Data Jockey leads to higher utilization of the upper storage tiers while minimizing the programming effort of data movement compared to human involved, per-domain ad-hoc data management scripts.

I. Introduction

Scientific workflows have become increasingly sophisticated with the growing emphasis on data analysis to keep up with data growth from experiments, observations and simulations [1]. This growth has given rise to several performance and data management challenges for scientific workflows running in HPC environments. To address such challenges, modern HPC storage infrastructure is evolving to have deeper storage hierarchies. In particular, a tiered storage architecture is gaining more interest in large-scale HPC deployments, and such a trend is expected to continue for the next decade [2], [3]. While the tiered storage architecture has been around for a long time to overcome limitations of monolithic storage architectures, it is gaining renewed interest as new storage technologies, e.g., SSDs, diversify and enrich the hierarchy.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

The combined impact of complex data pipelines (workflows) and deeper storage hierarchies (multiple storage tiers) has made data management on modern HPC centers a complex and challenging task. The current data management practices, which rely heavily on ad-hoc, manual data migrations are no longer feasible to manage vast amounts of scientific datasets across multiple storage tiers. Instead, providing a simpler storage abstraction that hides the underlying storage architecture is necessary to empower scientists to focus on the scientific discovery process.

Despite a few existing solutions, data management tasks are still largely relegated to the users. For instance, while users can employ scientific workflow managers [4] [5] [6], they typically cannot cope with multiple storage tiers. Similarly, existing tiered storage management mechanisms [7] and automatic data management systems [8] are not suitable for large-scale HPC data centers, because they are not designed to accommodate the batch-oriented, workflow-driven nature of HPC applications, and thus cannot readily coordinate with existing system components (i.e., job scheduler, file systems, archival, and workflow management systems). Furthermore, such systems aim to facilitate system administrator's tasks (i.e., performing system-wide data migrations) and do not provide an intuitive abstraction for end-users to orchestrate data migrations conforming to their individual needs.

To fill this gap, we present **Data Jockey**, a user-driven workflow-aware data management system for multi-tiered storage systems in batch-oriented HPC environments. Our goal is to automate the task of orchestrating bulk data movement and placement of datasets consumed by scientific workflows. Data Jockey is deployed as an HPC center-wide consolidated data management service that intelligently migrates datasets and manages storage capacities as required by target scientific workflows.

To support such automation of data management, Data Jockey presents a new data management scheme called "goal driven data management" that provides a declarative way of defining data movement under user workflows. In particular, this scheme reduces the burden of programming data movement by managing the dynamic state of user data and using that to automate source to destination data movement. With this scheme, Data Jockey is designed to have two planes

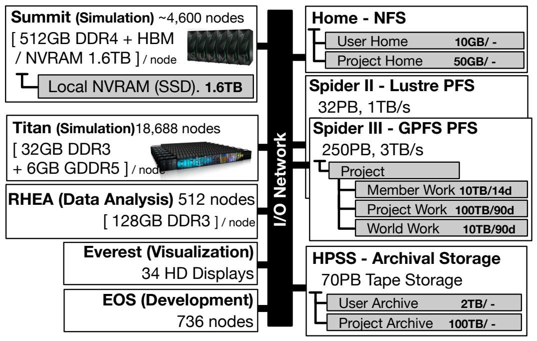


Fig. 1: **Multi-tiered storage in HPC:** Users in HPC centers have to manage data across several storage tiers or locations. Above is an example of Oak Ridge Leadership Computing Facility (OLCF) [9] where users have to manage data across multiple physical storage systems but also multiple logical storage tiers with different policies being applied [10].

that separate control and data. The control plane implements the data orchestration scheme as a centralized service that generates small control tasks about data movement, while the data plane transforms such control into bulk data movement. In between, a resource abstraction layer provides a unified interface to bridge control over heterogeneous data plane resources.

Contributions: The main contribution of this work is the design and implementation of Data Jockey that, at the core, presents a new "goal driven data management" scheme. In this work, we describe the concept, design, and architecture of this scheme and demonstrate its impact on a modern HPC data center. An analysis that quantifies the complexity of day to day data management is provided where we present how much complexity is reduced by our method. Further, we have deployed a prototype implementation of Data Jockey that implements the key features of our design in a real HPC environment and demonstrated its feasibility as a center-wide data management system that can be used in modern HPC data centers. Data Jockey is capable of reducing 85.7% to 99.9% of the programming complexity of data management, enabling users to better utilize the higher-performing, upper storage tiers.

II. Background and Motivation

A. Multi-tiered Storage in HPC

Multi-tiered storage is gaining interest as new storage technologies such as flash-based SSDs diversify the storage hierarchy. For example, the Oak Ridge Leadership Computing Facility (OLCF) [9] will have four storage tiers, namely the NVM-based burst buffer, the disk-based Spider III GPFS parallel file system (PFS) along with its predecessor Spider II Lustre PFS temporarily available before retiring, and the disk/tape-based HPSS archival storage, all to accommodate demanding storage needs from supercomputers including the 200 petaflop Summit system [2] (No. 1 in the Top500 list). These storage tiers support diverse I/O requirements such as the need to absorb high-speed, bursty and transient checkpoints, medium-term data analysis and long-term data retention. Other facilities have introduced yet another layer, the campaign storage, between the PFS and the archive, to support extended data analysis. Such a trend intensifies the already complicated storage hierarchy, e.g., currently four physical and twelve logical storage tiers in OLCF (Figure 1), placing an adverse impact on day-to-day data management tasks of users.

Simulation output: Each storage tier imposes a trade-off between performance and capacity. Particularly, large-scale simulations running on a supercomputer require a high-performance storage tier (e.g., burst buffer) that can rapidly absorb simulation output and minimize I/O jitter. Also, such output datasets should be migrated in a timely fashion to a higher capacity storage tier (e.g., PFS) to reclaim the scarce capacity of the high-performance tier. For further longer-term data retention, these datasets should survive various capacity constraints, including file system quota and data purge cycles.

Analysis input: A target storage tier of a data analysis job (e.g., burst buffer) may differ from a tier where the input dataset already resides (e.g., archival storage). This implies that an extra process of preparing the input data is likely to become required, which can be costly for large datasets. Moreover, conflicts between performance requirements of users and a limited capacity of each storage tier exacerbate the complexity of such data preparation processes.

Scientific workflows: Scientific workflows deal with large amounts of data that have different I/O characteristics. Managing such large amounts of data across a multi-tiered storage system requires workflow specific data management. For instance, users may want to move the input datasets to a fast storage tier like burst buffer, store and delete intermediate data as the workflow executes, and copy final outputs to a persistent store for further analysis. Currently, users explicitly move these datasets between the storage tiers before, after and during the execution of a workflow.

III. Data Jockey

A. Goals

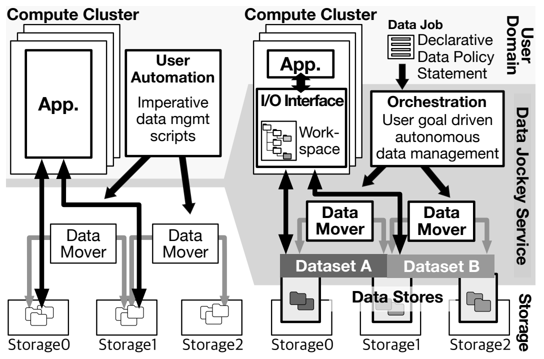
Data Jockey is designed and implemented to automate manual data management tasks for HPC users that have large datasets and complex workflows.

Batch-oriented, workflow-aware: Data Jockey targets batch-oriented HPC use cases where bulk data movement and access are driven by scientific workflows. Such workflows can either be an ad-hoc sequence of jobs submitted by a human or a graph of tasks that a workflow management system submits. To achieve this target, Data Jockey coordinates with existing job schedulers and workflow managers.

Consolidated data management: Data Jockey aims to be a consolidated data management system that accommodates center-wide data management needs. Data Jockey should deal with the heterogeneity of scientific workflows and the resources since data management can vary between scientific

TABLE I: Data Jockey Overview

Challenge	Approach	Related
Catalog management	A unified catalog that tracks datasets and their copies as well as associated data movers and data stores	Resource management
Storage tier selection	User policy-driven dynamic selection of storage tiers transparent to users	Data policy support
Data movement orchestration	A generic orchestration engine that automatically programs low-level data movement from a high-level declarative description of a desired "goal-state" of data	Goal-driven data management
Capacity management	Data movement scheduler that automatically moves datasets based on its knowledge of governing workflows	Workflow-aware scheduling
Coordinate data movement and access	Transparent systematic guarantees of non-interference designed to coordinate bulk data movement and bulk data access	I/O interface layer



a) Users handle the complexity of data management

b) Data Jockey alleviates the complexity of data management

Fig. 2: Architectural Overview

domains. Also, Data Jockey should be scalable enough to accommodate data orchestration requests from many users.

User-centric: Due to the user-centric nature of scientific workflows, Data Jockey aims to provide a user-centric data management scheme, which differs from existing policy-driven automation systems [11], [12]. While other policy-driven automation systems are mainly for administrators that implement few system-wide policies, Data Jockey aims to serve many data requirements (policies) directly from the users.

Integrates with HPC data centers: As a data management system, Data Jockey aims to integrate with existing HPC components such as schedulers, workflow managers, file systems, storage systems, and various data movers. Instead of replacing such components, Data Jockey focuses on implementing minimal wrappers to make such resources accessible to the upper orchestration components.

B. Overview

Data Jockey is a consolidated data management system that solves data management challenges (Table I). This consolidation replaces ad-hoc workflow specific user automation scripts related to data movement orchestration that users have to write otherwise. Figure 2 depicts the high-level architectural overview of Data Jockey.

Data job submission: Users or workflow managers submit "data jobs," which are similar to "compute jobs" (i.e., jobs for a batch job scheduler). These data jobs specify high-level goal states (i.e., pre-staged, persisted, replicated, safe) of datasets as a scientific workflow progresses or even beyond a single workflow. While batch jobs are submitted to the job scheduler service (i.e., PBS, Torque/Moab, LSF, Slurm), data jobs are submitted to Data Jockey through an independent data job queue manager that interacts with the job scheduler.

Data orchestration pipeline: Data Jockey implements a new data orchestration scheme called "goal driven data management" when handling each data job submitted into the data job queue. By going through multiple stages, high-level data requirements described in the data jobs are translated into low-level data movements. While doing so, Data Jockey performs dataset shuffling based on the future timeline of data usage. Soon to be used datasets are promoted to a closer storage tier while datasets to be used later are demoted to a further storage tier.

Resource manager: Data Jockey actively tracks and modifies the state of user data and the underlying storage infrastructure (data storage and movers) by maintaining an information base as well as implementing an API that provides a unified interface towards these resources. We introduce high-level abstractions such as data jobs, datasets, data movers and data storage (shown in Figure 2) to generalize various data management interactions.

I/O interface: To coordinate I/O and data movement, Data Jockey introduces an I/O interface layer that implements an isolated virtual namespace local to each compute job. This I/O interface prevents collision between data movement and access while providing redirection to the appropriate tier.

C. Using DataJockey (Use-cases)

*1) **Automatic data staging and de-staging**:* The primary use-case of Data Jockey is to automate data staging and de-staging in the context of multi-tiered storage systems. Running a job atop a multi-tiered storage system requires some form of capacity management for datasets due to the capacity limitations usually enforced on tiers that are directly available to applications (i.e., tier one). Manual data movement might be preferable and straightforward for rigid shallow storage hierarchies but quickly becomes unmanageable when the number of storage tiers and the number of datasets increase. In-house automation of such data movements is prone to errors since users are not trained software engineers, and this eventually leads to human intervention.

Data Jockey provides automatic dataset promotion based on priority and coordinates with the job scheduler. Victim selection and eviction are automated to prepare space for input or output datasets, all accompanied with transparent out-of-band bulk data movement. For users, such automation comes at the minimal cost of a one-line directive in-line to the job script per dataset.

*2) **Automatic data preservation:*** With Data Jockey, users can specify policies on user datasets to preserve unused datasets in the face of unexpected events such as dataset purge cycles or job failures. For example, under circumstances where the human in the loop is absent (i.e., out for a conference), Data Jockey provides an integrated solution for automatic data preservation. With a specified timeout of inactivity on a dataset, datasets can be left on higher tiers until it is mandatory for the dataset to be moved into safer storage. Additionally, target destinations can be arbitrary where users can set policies of multiple safe locations to preserve datasets.

*3) **Multi-cluster workflow:*** Another compelling use-case is to automate bulk data movement in the context of multi-job workflows that span across multiple compute clusters or even different HPC facilities (Figure 1). For such use, Data Jockey is used as a shared data supply substrate that is aware of the use of multiple storage tiers. Data Jockey employs a resource management scheme that manages storage systems and data transport methods as a network of data stores and data movers. This network is shared by multiple end-points (i.e., multiple compute clusters or storage archives) where storage hierarchies are dynamically computed based on proximity to such end-points.

With the same directives in the job scripts, Data Jockey provides automatic point-to-point dataset movements tightly integrated with the automated capacity management mentioned in Section III-C1. For example, datasets move closer to the target storage tier available to the target cluster as the associated job is soon to be executed. In the process, lower priority datasets are automatically pushed to alternative storage tiers according to the storage hierarchy dynamically computed with respect to their target clusters.

In the following sections, we discuss how the aforementioned use-cases are realized using Data Jockey's design and implementation.

IV. Design

A. Abstractions for data management

*1) **Data job:*** The primary purpose of a data job is to specify and trigger bulk data movement that prepares or post-handles datasets for/from compute jobs (i.e., jobs for schedulers such as PBS, Torque/Moab, LSF and Slurm). Data jobs are defined to have one-to-one relationships with compute jobs and are submitted in parallel, triggering data movements before and after the execution of the associated compute job. By doing so, a data job deports in-line ad-hoc data staging and de-staging scripts to outside of the compute job context and replaces such scripts with directives for extended data orchestration that are not limited to data staging and de-staging. To support such automation, Data Jockey provides a stand-alone orchestration engine that interacts with the job scheduler to coordinate data job and compute job execution. When replacing the in-line scripts, data jobs are specified using a declarative specification method (detailed in Section V-A) to simplify data layout operations. In a data job, users describe an array of <dataset, goal state, policy, mountpoint> tuples (detailed in the following sections).

*2) **Dataset and their replicas:*** Rather than relying on storage system abstractions such as files or objects, Data Jockey introduces a high-level abstraction called a "dataset" that is defined as a collection of dataset replicas placed in the storage hierarchy. These replicas are a collection of individual files or objects. This coarsely defined abstraction enabled us to mitigate the cognitive burden of handling tens of thousands or millions of files which does not necessarily reflect the user view of data, enabling us to reflect user data management workflows or policies that are often at a much higher level than files. Within a dataset, each replica can have different attributes (i.e., location, stripe size, state) but the data itself (collection of files) remains identical. These replicas are organized to have 'master' and 'slave' relationships where the 'master' replica is handled to be the primary replica.

*3) **Goal state of a dataset:*** In a data job, the goal state of a dataset is assigned to the datasets, representing the desired physical shape of the datasets requested by the users. By providing such information, users can control where and how datasets are prepared for a compute job. To capture such information, we define states as a set of attributes (i.e., location, stripe size, encryption, compression) used to configure replicas, canonically named (buffered, staged, archived). For example, a "staged" dataset state ready on a parallel file system (i.e., Lustre) with a particular stripe size can be materialized with two key-value pairs of <location, lustre> and <stripe size, 128> under the name "staged". At system design time, admins (or power users) are expected to define such states, where users refer the states by name upon job submission. When submitted via data jobs, Data Jockey supplies such attributes into low-level "data actuators (data stores and data movers)" that are used to prepare datasets.

*4) **Dataset policies to handle events:*** In data jobs, dataset policies are also assigned to the datasets. Policies are used to handle events like job failures, timeouts, and cancellations. Policies are structured like a try-catch (or except) block seen in modern programming languages. There is a primary directive (try block) being executed, but when an exception happens, exception blocks (catch / except) are executed. Such conditional structure of policies is similar to rule-based action policies implemented by other systems. However, Data Jockey differs on how actions are described. To limit the complexity of programming actions of such rules, goal states are used in place of the actions. When exceptions happen, actions are parsed as "apply state X to dataset Y" instead of performing a sequence of actions. Similar to goal states, policies are also expected to be defined at system design time, later referred by name in data jobs with canonical names like "default" or "auto-backup".

*5) **Mountpoints for dataset consumption:*** Mountpoints are locations in the application storage namespace (i.e., filesystem) where applications can consume the designated datasets. Such mountpoints are passed to the I/O redirection component that

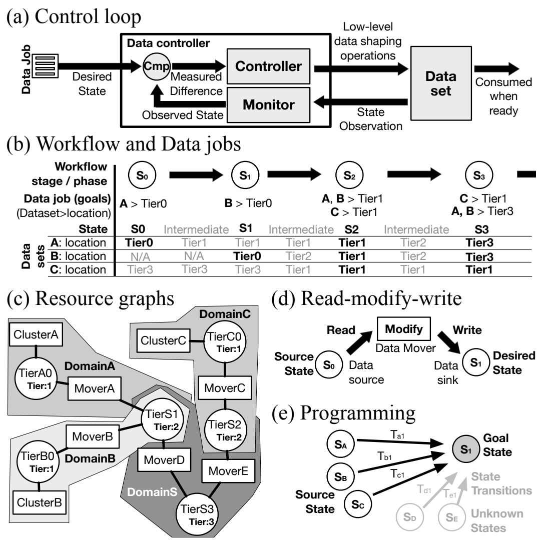


Fig. 3: Goal driven data management scheme

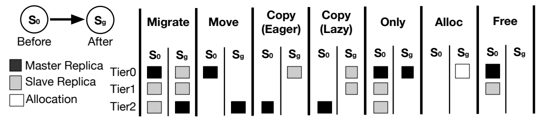


Fig. 4: State transition methods (replica handling)

translates access to mountpoints on the physical location of the datasets. This I/O redirection component is configured after a data job prepares a replica with the desired state pinned in place. Later, after compute jobs have finished executing, datasets are unmounted, redirection component is torn down, and datasets are unpinned.

*6) **Data stores and data movers**:* In Data Jockey, data stores are data sources and sinks for datasets, and data movers are the data pipes in between. Data stores represent logical-physical (also geographical) locations that are regions (subtree) of an underlying storage system storing user datasets, typically implemented using a "container" abstraction (i.e., directories in POSIX) provided by the underlying storage system. Data movers represent data moving methods that exist between or within storage systems that can be a simple UNIX "cp" command or a sophisticated parallel copy method or a file transfer method between distant geographic locations.

B. Goal Driven Data Management

Figure 3-(a) shows the concept of the "goal driven data management" scheme proposed in this work. The main idea of this scheme is to maintain a control loop that modifies the state of user datasets laid out across the storage infrastructure. The goal of this control loop is to ensure that a replica that matches the user goal exists in the system when requested.

Upon executing a compute job, this control loop accepts the desired state (i.e., desired storage tier) of datasets (data job) as the reference input and compares it to the current state of the replicas Data Jockey already has. Such comparison is performed by comparing metadata attributes that represent the state at the level of replicas (collection of files). If there is a difference, this loop computes the necessary low-level movements required to prepare a replica that has the desired properties. Afterward, the prepared replica is consumed by the associated computer job. This concept of a control loop is inspired by cluster management systems such as Kubernetes [13], Ansible [14] and Puppet [15]. In the case of Data Jockey, such control is applied by managing the state of user datasets instead of infrastructure.

*1) **Scientific workflows and data jobs**:* In the context of scientific workflows, the control loop accepts multiple data jobs that describe the state of datasets required by each compute job or workflow stage as the workflow progresses (Figure 3-(b)). While data jobs only describe the desired state of datasets required during the execution of a compute job, the intermediate states are actively tracked by Data Jockey. For Data Jockey, the sequence of data jobs is a constantly shifting system goal that the control loop has to reconcile. For this sequence of data jobs, Data Jockey provides the necessary synchronization with the associated compute jobs.

*2) **Dataset state interpolation**:* To support such dynamic control on user datasets, Data Jockey maintains a catalog of the required data management resources in the form of a graph (Figure 3-(c)). In particular, Data Jockey maintains information about the topology of data management resources such as data stores and data movers, each translated into a vertex (data store) and an edge (data movers). Data stores represent individual storage tiers (i.e., file systems, archival storage) while data movers represent the data paths in between (i.e., POSIX cp tool, parallel copy, archival storage backup tool). Here, Data Jockey considers 'location' as a primary attribute for a dataset and uses the graph representation to compute the necessary steps (multiple source-to-destination copies) to reconcile the state of a dataset. Also, Data Jockey dynamically extracts the storage hierarchy from the graph and applies it to data movement.

*3) **Dynamic storage hierarchy**:* Data Jockey dynamically extracts the storage hierarchies based on a coarsely grained 'tier number' given by the users that hint the orientation of the data stores, influencing data movement within the resource graph (Figure 3-(c)). In the resource graph, storage tiers with higher tier numbers are peaks of a landscape where user datasets either climb towards a higher tier (promote) or descend to a lower tier (demote), all using the data pathways (edges) between the data stores (vertices). This design is to support multiple compute clusters in an HPC center (multiple peaks) that are designed for specific tasks (e.g., compute, simulation, analytics) since storage hierarchies can vary from the perspective of two different clusters due to the diversification of storage tiers (i.e., SSD based performance tier dedicated to a certain cluster).

*4) **Shaping datasets from state to state**:* When enforcing state transformation of each dataset, Data Jockey uses a sequence of read-modify-write of a replica (Figure 3-(d)). If a replica of a desired set of properties does not exist, Data Jockey reads a replica (read from a storage tier or a compute job) and creates (write) a new replica with the desired set of properties (modify). During these replica level read-modify-write operations, Data Jockey guarantees that no other I/O accesses are performed on the replicas involved to maintain data integrity. Data Jockey achieves this by owning the replicas in a managed space where no third-party I/O is allowed except for the ones that are explicitly allowed. Since such a read-modify-write cycle always introduces a new replica, Data Jockey implements several methods that control the number of replicas in the system (Figure 4). Stale or excessive replicas produced during the process are invalidated and garbage collected (i.e., a new allocate invalidates all existing replicas).

*5) **Impact on programming**:* Goal driven data management significantly reduces the amount of information required from the users in order to automate data orchestration. Figure 3-(e) depicts a simple model that quantifies this impact. With Data Jockey, it is sufficient for users to specify the destination state since the transitions between the current state is automatically inferred. Without Data Jockey, assuming the primary attribute of a state is a location, a user might have to prepare multiple source (source state) to destination (destination state) copy statements (state transitions) for all known data paths and implement a mechanism that is capable of selecting proper data paths based on the current location of a particular dataset.

V. Implementation Details

A. Data job specification

*1) **Multi-level abstraction**:* Data job specifications are structured to have multiple levels of abstractions. This structure is to limit the complexities being exposed to the end-users but also to be powerful enough for power users. Data job specifications are composed using two key components. One is the "library" and the other is the "directives" used in job submission scripts. Libraries define the vocabulary of data jobs, and directives use the vocabulary to compose and submit a data job.

*2) **System administrators, users and power users**:* At system design time, system administrators are expected to define the library of possible dataset goal states and policies when they setup available data stores and data movers. Later in normal operations, users interact with Data Jockey by referring to this pre-defined library of goal states and policies by their name. In doing so, the pre-defined library serves as the system default. Power users may choose to override or customize existing elements in the library. In general, average users may end up writing one-line directives that refer to entities in the library. Advanced users would customize the library.

*3) **Library**:* Libraries provide the vocabulary of data management operations, defining the goal states and the dataset policies. Data Jockey uses YAML [16] to define a library, exploiting the hierarchical structure of YAML that scopes key-value pairs (Figure 5-upper). Under the block "policy", a "default" dataset policy is defined that handles three exceptions. Also, under the block "placement", four possible goal states

```
# library.yml
policy:
 default:
  main: { default: 'ready', timeout: 13d, max_replica: 3 }
  on_evict: { apply: 'safe' }
  on_timeout: { apply: 'safe' }
  on_error: { apply: 'persist' }

placement:
 burst_sink: { storage: ['bb'], method: 'allocate' }
 ready: { storage: ['bb', 'pfs_scratch'], method: 'clone' }
 persist: { storage: ['pfs_project'], method: 'migrate' }
 safe: { storage: ['hpss'], method: 'migrate' }
```

```
#!/bin/bash
# simulation.pbs
#PBS -A pjt000 -N test -lwalltime:1:00:00,nodes=1500

#DJ "ds://pjt000/dataset0,./analysis/input,analysis"
#DJ "ds://pjt000/burst0,./output,dump"

mpirun -n 1500 ./a.out     # Executing the main application
```

Fig. 5: Data job specification

are defined. Each of these policies or goal states (placements), are then defined with a set of key-value pairs accepted by the orchestration engine. Libraries have a hierarchical structure where higher level entities are built by composing multiple low-level constructs. A "policy" is built on top of multiple relevant "placements" (goal states) referred within the catch blocks. Also, entities in the libraries have a layered structure where each key-value pair is an override of the defaults.

*4) **Job script in-line directives**:* With the vocabulary defined in the library, users submit data jobs with a set of directives embedded in a job script. Data Jockey provides a job submission wrapper that submits a data job from the directives while submitting the main compute job to the job scheduler (detailed in Section V-B). These directives are used to produce an array of tuples, consisting of 1) a reference to a dataset, 2) desired namespace mount location, 3) a reference to the goal placement (goal state), 4) a reference to a placement policy, and 5) a list of optional constraints such as the deadline for the data job to reach the "goal state", making datasets available to the compute job.

The lower part of Figure 5 is an example of such directives used in a job script formatted in PBS (Portable Batch System). The directives are embedded with the `# DJ` prefix while other implicit settings are imported from the environment. In the example, Data Jockey prepares an input dataset at path ./input using the 'analysis' goal state in the current work directory. Also, an empty dataset is created to dump output at a path ./output using the 'dump' goal state. Preparation of two such datasets, one existing and one to be created is achieved with two lines of such directives.

In-line embedding of directives is not limited to one job script format. Support for other types of job scripts is a matter of having another job submit wrapper tool for another scheduler.

B. Scheduler Integration

Data Jockey loosely integrates with job schedulers in order to coordinate the execution of data jobs and their associated compute jobs. When integrated with Data Jockey, job schedulers are not aware of data jobs. Data Jockey behaves as if a user submits a job and monitors the queue for further control. To achieve this, Data Jockey provides a job submit wrapper tool and a job queue control service. We chose such loose integration as a default to enable Data Jockey to be integrated with broad types of job schedulers.

*1) **Job submit wrapper tool**:* Job submit wrapper tools are provided to perform parallel submission of compute jobs and the embedded data jobs. Such tools are provided as a wrapper CLI tool that accepts `stdin` or filenames of job scripts (Figure 5). Data jobs are extracted and submitted from in-line Data Jockey directives (Section V-A4) and the compute job is submitted to the job queue control service.

*2) **Job queue control service**:* When data jobs and compute jobs are submitted, the job queue control service ensures that the execution of data jobs do not overlap with the execution of compute jobs. This service ensures compute jobs are not executed until the data jobs finish preparing the requested datasets. Such control is achieved by monitoring the progress of data jobs and also monitoring and controlling the status of the compute job using integration methods (i.e., wrapping job scheduler CLI tools).

C. Data Orchestration

The data orchestration pipeline implements the proposed "goal driven data management" scheme. This pipeline implements the core part of the control loop.

*1) **Data placement resolution**:* Goal state of a dataset for dataset requests are represented as "placement" components implemented as a set of attributes that describe the goal state of a particular dataset. Such attributes include destination storage (i.e., burst buffer, parallel file system, archival) and methods (Figure 4). "Goal states" placements are canonically named by users or administrators to represent the desired state of datasets (i.e., ready, persist, safe). These canonical names form a user-friendly vocabulary when defining a higher order "policy" component that enables users to define a goal for the system to pursue. With such policies, users state a primary goal placement where there are optional goals that should be pursued upon system exceptions such as job failures, evictions, timeouts, and errors.

*2) **Data movement planning**:* After the placement for a dataset is resolved, a data placement request is issued to a data movement planning stage. The target of the data movement planner is to place the replica to the designated storage tier even if it requires evicting existing replicas. To facilitate the decision of what stays and what leaves, Data Jockey uses the priority property which was given to the data jobs. With the current implementation, the submission wall clock time is used as the priority.

Tight integration with job schedulers (dataset and data job aware scheduling) are left for our future work.

Phase1 - Nearest replica and promote location: For each dataset request, the planner asks the resource manager for the nearest replica (in the resource graph) in the dataset that can reach the goal. Next, the tier to get closer to the goal tier is inferred by the resource manager and the request is placed in a waiting queue on the tier.

Phase2-A - Placement requests: Then, the storage planner iterates through the storage tiers from top to bottom to process the placement request assigned to each tier. In this process, the planner attempts to place the replicas in the tier but also tries to find victim replicas it can 'demote'. Here, requests that can fit without evictions are marked actionable.

Phase2-B - Handling evictions: The requests that require demotions are temporarily marked dependent on the victims. For these victims, the planner gets a demotion location from the resource manager. When victims are placed to a demotion storage tier, the request inherits the priority of the root placement request that caused the eviction. This action can be chained throughout multiple storage tiers and are triggered to be 'actionable' when the demotion of the furthest victim can be fulfilled (enough quota), otherwise canceled.

Phase3 - Reap actionable: After the planner determines the fate (actionable, non-actionable) of each placement request, the planner reaps the chain of actionable requests starting from the bottom tier to up. Here, a convoy of data movements is issued starting when an evicted victim is finally marked actionable. For example, simple space allocation for a replica on the top tier that required a cascade eviction down two tiers would result in three actions where the bottom tier data movements are fulfilled first, followed by the upper tier data movement and finally the actual allocation operation the user wanted. In the sequence of actions, the planner guarantees that the storage quota consumed by the convoy is locked from subsequent planner iterations.

*3) **Resource manager support**:* To support the planner, the resource manager maintains the graph-based dynamic storage hierarchy mentioned in Section IV-B3. Queries such as finding the nearest replica towards a goal storage tier is a matter of evaluating the cost (e.g., bandwidth of edges) of shortest paths, and finding a data movement plan towards a goal location (i.e., promotion) is a matter of translating the paths to a series of source-to-destination data movement steps.

In this graph, data stores are labeled with tier numbers that define the altitude of their position in the storage hierarchy, influencing replica demotion. For demotion, a lower altitude neighbor that has the shortest path towards the goal location is selected. Since the planner performs the scan starting from above (higher tiers), the demoted replicas generally flow towards lower altitude storage (valleys in the graph) when evicted by higher priority replicas.

*4) **Data movement scheduling execution**:* After the planning stage, the data movement execution stage dispatches the queued actionable data movements. Data Jockey implements a low-level scheduler that selects eligible data movements and assigns them to the resource agents that are available. The progress of these data movements are then tracked and

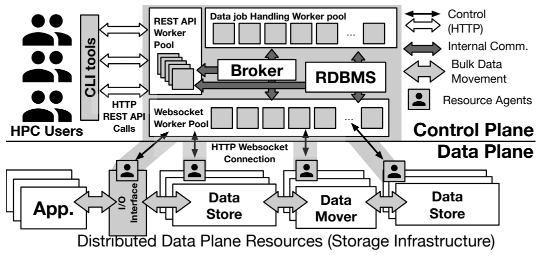


Fig. 6: Data Jockey implementation

managed. It is guaranteed that the progress falls into discrete atomic states such as 'queued,' 'pending,' 'error' and 'done'. Underneath, resource agents encapsulate the invalid states by performing automatic cleanup. This stage ensures that low-level data movements that have dependencies adhere to the correct sequence by honoring the dependency chain (convoy) assigned by the planner.

*5) **I/O session**:* After the data preparation stage is finished, the replica is pinned and an I/O session is opened to give access to the application. Within the I/O session window, a special resource agent instance that is run in-situ of the context of a compute job configures the I/O interface layer to provide access to the application. Currently, Data Jockey uses symbolic links to set up an I/O path towards a managed replica but can be extended with sophisticated I/O interposing mechanisms.

D. Data Jockey Implementation

Data Jockey was implemented to have a control plane, a data plane and a set of tools to provide a user interface (Figure 6). Data Jockey was implemented with approximately 15K lines of Python 3.6 [17] code that leverage a message broker (RabbitMQ 3.6 [18]) for internal communications and an RDBMS for metadata persistence (MariaDB 5.5.56 [19]). For the control plane's external communication with the resource agents and CLI tools, Data Jockey uses HTTP based REST APIs and WebSockets. Job scheduler integration was done with Torque 6.0.2 / Moab 9.1.1.

*1) **Control plane - Web application**:* The control plane implements data job processing, automatic replica placement, and orchestration. To interact with the users, the control plane exposes multiple REST API [20] endpoints that enable users to manipulate the state of data jobs, datasets, and resource agents. Manipulation of certain resources via the API (i.e., data job submission) triggers a cascade of operations backed by multiple stages of worker processes that pick up the necessary state to handle a certain stage, eventually leading to the issuing of low-level operations towards the data plane.

*2) **Data plane - Resource agents**:* Resource agents are agent daemon processes that form the communication backbone of the data plane. When deployed on a node, agents announce the capability of a particular node (i.e., mountpoints, data movement tools) to the control plane via a WebSocket connection and subscribe for low-level data management tasks. Within the control plane, such announced capabilities are used to construct a resource graph by connecting data stores and data movers based on available interfaces (POSIX, object store or other) and reachability domains (network).

*3) **Concurrency model and coordination**:* Data Jockey has been implemented in an asynchronous event-driven architecture backed by a shared pool of stateless worker processes and a common persistent layer. In Data Jockey, each data object related to a data job is implemented as a state machine where the states are persisted as database records. State manipulations are handled by any process available in the pool when an event occurs. This architecture was implemented using the RDBMS to persist state manipulations while using the message broker to maintain and coordinate the worker pool. Data Jockey heavily relies on the transactional guarantees of the RDBMS to make consistent updates of flags and state fields of multiple objects.

*4) **Fault tolerance and scalability**:* The architecture described in Section V-D3 makes Data Jockey relatively easy to scale and be tolerant to failures. When worker processes fail to handle an event, the message broker guarantees a retry while the atomicity of a state manipulation is guaranteed by the RDBMS backend. Also, scaling the control plane is a matter of increasing the number of workers in the pool. In this design, the backend RDBMS is the bottleneck, but this issue can be addressed with database sharding techniques. Data Jockey relies on a project-oriented resource model that can better support such partitioning techniques.

VI. Evaluation

In this section, we present the evaluation results for the Data Jockey, seeking to answer the following questions.

- What is the impact of Data Jockey in terms of user experience?
- Can Data Jockey be deployed as a center-wide data management service for HPC facilities?
- How well does Data Jockey integrate into HPC environments?

To answer these questions, we evaluated a prototype of Data Jockey that implements the key features described in Section V. Our prototype has been deployed and evaluated in two separate environments, a small-scale testbed, and a mid-scale HPC production environment. We have obtained 18 months worth of operational scheduler logs from OLCF and used the analysis of these logs to guide our evaluations.

A. Quantifying the Impact of Data Jockey on User Experience

We measured Data Jockey's usability on how much it reduces the complexity of data movements from a user perspective. Considering system states and transition paths as vertices and edges in a state diagram (Figure 3-(e)), we measured the complexity of a workflow as the total number of vertices/edges that users need to account. In a conventional script, such elements translate into branches of if-else control-statements. For Data Jockey, these elements translate into object definitions within the data job specification.

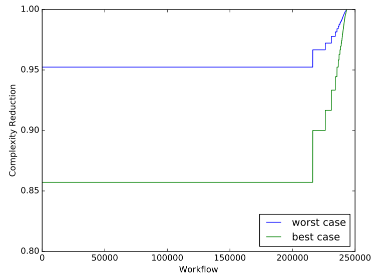


Fig. 7: Complexity reduction of Data Jockey on workflows in the OLCF operational log

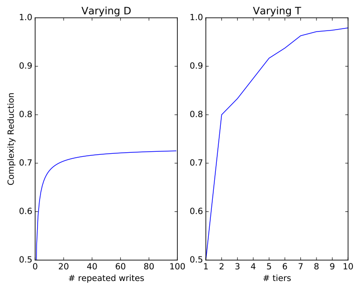


Fig. 8: Complexity reduction of Data Jockey on workflows by varying parameters

For a workflow on a multi-tiered storage system with T storage tiers, assume the workflow produces datasets with the same write pattern D times, each time generating a single aggregate dataset. In the system, T storage tiers are connected by V paths; datasets can be transferred from one tier to another by the data movers in between; in the data-movement process, E failures may occur on storage tiers or data movers before a data transfer succeeds.

Thus, when a workflow owner designs data movements on a multi-tiered storage system, in the worst case, $D \times T \times V \times E$ if-else design cases should be considered. In the best case, the user can specify tiers and paths and we can consider a smaller set of design cases: $D \times E'$, where E' is the error on the specified tiers and data movers. Thus, with the use of Data Jockey, the system-side complexity is reduced ($V \times T$). To measure the efficiency of Data Jockey, we estimated the complexity reduction by: $1 - \frac{D \times E}{D \times T \times V \times E}$ for the worst case and $1 - \frac{D \times E'}{D \times E' \times V \times T}$ for the best case.

We analyzed the complexity reduction due to Data Jockey based on OLCF operational logs collected from January 2017 to August 2018. In this study, we considered the jobs under the same project ID as a workflow. The OLCF data includes 243,265 workflows, each workflow having 1—158,582 computational jobs and one analytic job; each computational/analytic job produced a single dataset. Figure 7 reports the results. It suggests that for workflows in the OLCF storage systems, Data Jockey attained 85.7%—99.9% complexity reduction, with 216,151 workflows (88.85%) having a single computational job and a single analytic job. Specifically, for these simple workflows, Data Jockey achieved the lowest complexity reduction with 85.7% for the best case and 95.1% for the worst case, respectively; with the increase in numbers of jobs, the complexity reduction increased.

TABLE II: Performance goals

1) Job processing performance (unit: jobs/min)

	Mean	Max	Goal
Arrival rate	2.29	423	-
Start rate	2.1	36	>36
Completion rate	1.97	290	-

2) In-flight jobs (unit: jobs)

	Mean	Max	Goal
Waiting	1.15	300	>300
Running	1.03	29	>29

Moreover, to understand the complexity reduction due to Data Jockey in different settings, we studied the performance by varying the number of repeated writes (D) and the number of storage tiers (T). Specifically, we simulated the behaviors of users and Data Jockey based on these varying parameters. At the start point, we assumed that the workflow produces a single dataset running on a single-tier system .

Figure 8 reports the results. By varying D, Data Jockey attained the same complexity reduction ranging from 35.45% to 72.54%; by varying T, Data Jockey attained a reduction ranging from 35.45% to 99.21%. It suggests that when the system-side complexity grows (D, T), Data Jockey reduced more complexity from users. In the extreme cases (e.g., T =10 in Figure 8), Data Jockey can achieve $>$ 99% complexity reduction.

B. Feasibility as a Center-wide Data Management Service

Since data jobs are issued to prepare data for compute jobs and are executed in parallel, data job processing rate of Data Jockey should keep up with the compute job scheduler when deployed. For this reason, the performance goal for processing data jobs were set to exceed the maximum job start rate and the maximum number of in-flight jobs observed waiting or running (Table II).

For this evaluation, we set up a controlled testbed (Figure 9-(a)). In this testbed, we employed eight x86 nodes each equipped with one AMD EPYC 7351 16-core processor that is capable of 32 H/W threads (2.40Ghz 6MB Cache) and 128 GB of main memory. Compared to a real HPC environment, this environment lacks access to real storage tiers and bulk data movement used in production.

To load the control plane, we implemented a workload driver that can mimic the data job submission behavior under a fixed workflow depicted in Figure 9, (b) and (c). Under this workflow, the dataset placement planner acknowledges

In the simulations, we assumed that relative paths (V) and data movement errors (E) are dependent parameters and vary according to D and T on Weibull distributions [21].

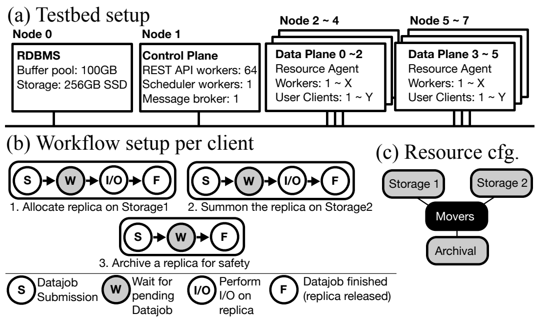


Fig. 9: Testbed setup

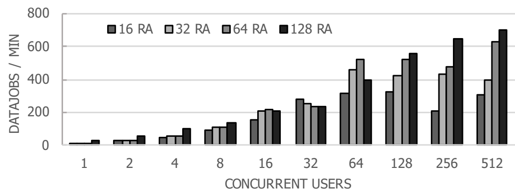


Fig. 10: Impact of concurrent users and resource agents: 16 to 128 RAs (resource agents)

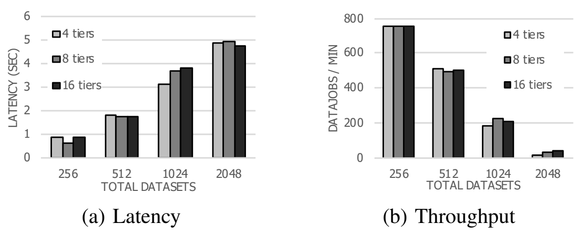


Fig. 11: Impact of the number of storage tiers and user datasets (256 concurrent users, 128 resource agents)

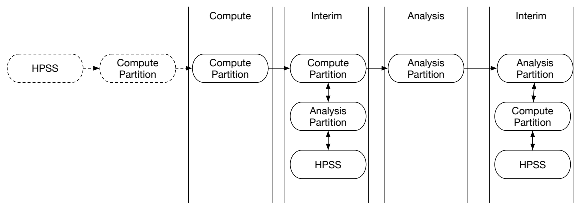


Fig. 12: Possible data paths that a dataset will follow during the course of the evaluation campaign.

the size of each replica but the resource agents were set up to move zero bytes of data even though they receive commands to allocate, move and delete entries. Also, we fixed the size of the replicas being allocated to have a constant flow of data jobs being processed.

Figure 9-(a) shows data job processing throughput of a single-node control plane under a varied number of users and resource agents (Figure 10). The control plane was able to scale up to approx 70 data jobs per minute but required more resource agents to achieve more. With enough resource agents (`128 RA`), the control plane was able to process more than 600 data jobs per minute. This data job processing rate was enough to accommodate the job burst (36 jobs per minute - Table II) of our load. The control plane was able to sustain such performance over 512 concurrent jobs which was also enough to accommodate a maximum burst of 300 pending computational jobs submitted to our clusters.

We also evaluated the performance of a single data movement planner instance mapped for a single project under a varied number of storage tiers and datasets per workflow iteration (Figure 11). The performance of a single data movement planner was mainly impacted by the total number of datasets with performance degradation down to 18 data jobs per minute with 2,048 datasets. We assumed 1,024 – 2,048 active datasets (collection of files) were enough for a group of users in a single project.

With the results, we conclude it is possible to deploy Data Jockey serving large HPC clusters like Titan [22] or Summit [2]. A single node deployment with a single database instance was enough to accommodate the peak load.

C. Feasibility of Integration with HPC environments

In order to gauge the feasibility of integrating Data Jockey into HPC environments, we conducted experiments utilizing synthetically generated workflows with Data Jockey and compared the results against the same workflows managed by a user-like entity. These workflows are based on the analysis of production scale jobs of two distinct supercomputers. A workflow consists of multiple phases where each phase consists of a set of computational jobs and a single analysis job. Every computational job requires an independent input dataset and produces an output dataset. These output datasets are utilized as the input datasets for the analysis job that in turn generates a finalized, resultant dataset. The next phase cannot commence until the analysis job of the current phase has completed. Therefore, there is an explicit job and data dependency among not just the set of jobs within each phase but between the phases as well. This dependency chain closely resembles observed data patterns at OLCF.

For the HPC environment, we used 16 nodes from the Rhea cluster at OLCF with 14 of the nodes serving as virtual computational nodes and 2 as analysis nodes. Each physical node operated as a set of virtual nodes with each node scaling to 1,334 and 256 virtual computational and analysis nodes, respectively. By utilizing job packing, we were capable of handling a large number of jobs allowing us to integrate Data Jockey with the PBS scheduler on Rhea. Since each job performed no actual computation and very little I/O, utilizing ftruncate to create the dataset, we do not believe that the virtual scaling introduced an issue. These nodes also functioned as resource agents to manage the movement of the datasets.

We utilized three storage tiers for this evaluation, an archival HPSS tier and two Lustre PFS tiers provided by Spider II. The compute and analysis elements do not possess a shared

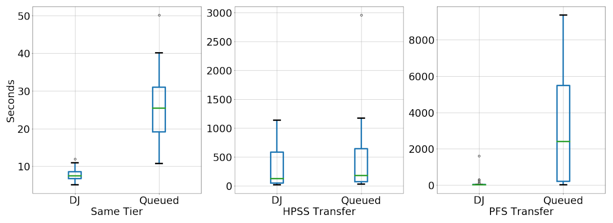


Fig. 13: Job wait time grouped by data transfer requirements.

PFS backplane, therefore, requiring that data be moved across to be accessible. Figure 12 illustrates the potential paths that a dataset may traverse during a workflow. Depending upon the size of the initial input dataset for a job, it may reside on either HPSS or the two PFSs with more massive datasets being initially located on HPSS. By the time of execution, Data Jockey has to relocate the dataset onto a PFS that the compute nodes can access. In the interim, between computation and analysis work, Data Jockey may migrate the input and output datasets among the three tiers as needed. However, Data Jockey must ensure that before the analysis job was released from the scheduler, it delivered the corresponding datasets onto the analysis PFS partition. After a workflow phase, the datasets were released and could be migrated back to the HPSS tier as Data Jockey deemed necessary for capacity reasons. Much of the data shuffling between the filesystems and HPSS is mandated by filesystem quota limits and purge policies. Work areas on Lustre are 10 to 100 TB and can be purged within 14 to 90 days. Without Data Jockey, users have to be cognizant about these policies or risk job stalls and even data loss.

Based on an analysis of the OLCF scheduler logs and per job I/O data from Lustre for three months, a driver script stochastically generated the workflows from two multivariate empirical CDFs, one for the computational stage and the other for the analysis stage. For the computational stages (first CDF), the number of concurrent jobs and per job node size, runtime, input dataset size, and output dataset size are randomly chosen. For the analysis stages (second CDF), only the node size, runtime, and output dataset size are generated. Further, for the analysis stages, the number of jobs is fixed at 1, and the computational stage's output determined the input datasets. Collectively, we have set the number of concurrent workflows to 4, twice the number typically witnessed in practice.

Beyond just demonstrating the feasibility and benefits of Data Jockey, the crux of the evaluation is proving that Data Jockey does not negatively impact the execution of the workflows. We executed several runs over multiple days resulting in over a hundred job runs and over twice that number of data jobs transferring data for the jobs. There were also several involuntary evictions as completed datasets were migrated from working to permanent storage on HPSS.

Figure 13 provides a breakdown of the latency introduced by Data Jockey. The results are grouped into three distinct boxplot graphs by the type of job and data transfer required for execution. The queued column provides the total time jobs spent waiting in the scheduler queue (excluding job dependency time), and the DJ column has the fraction of that queue time spent waiting on DJ decision-making and data transfer.

For datasets already existing on the proper tier where no data movement is required, Data Jockey adds no more than a mean of 7.9 seconds as shown in the leftmost graph. The time spent should provide the expected overhead of setup, tear-down, and decision-making for all Data Jockey jobs. Additionally, as those jobs are already queued, that latency may be entirely masked by the work of the scheduler.

The second and third graphs in Figure 13 illustrate the effects when data movement is required. Most of the time spent by Data Jockey is triggered by moving the datasets, and in the case where there are ample compute resources, as in the middle graph, the scheduler can release the job almost immediately. Where resources are constrained due to lack of nodes for analysis, as in the rightmost graph, we can see the entirety of Data Jockey masked by resource contention. Although Data Jockey extended the queue time for the middle graph, without Data Jockey the user would have had to execute the transfer before scheduling the job. The combined time of manually transferring the data and then submitting the job should be equal to or greater than with Data Jockey.

We implemented a user emulation script that simulates uncoordinated, multiple users executing their workflows. These users are ideal in that they perform constant monitoring of their jobs and data movements. The script is similar to the Data Jockey driver script; however, each workflow must manage their datasets as well as scheduling jobs. Lacking information about when their datasets will arrive, jobs cannot be submitted until after the datasets have been placed. Furthermore, workflows only transfer data to HPSS when there is an explicit need and not opportunistically, potentially introducing workflow stalls.

While conducting this evaluation, we experienced scheduler reservation issues that adversely affected the queue time of jobs emanating from Data Jockey. The mean job throughput was 3.7 jobs per hour for the user script and 2.9 jobs for Data Jockey. Closer examination revealed that this was due to the prolonged time that jobs were blocked by the scheduler. The mean times for dataset movement were essentially the same, but the queue time was 1,441 versus 468 seconds for the Data Jockey and the user scripts, respectively. Even though the overall result is subpar, an important caveat is that the user script is constantly monitoring and updating their jobs. Adding less than 5 minutes of user "think" time per job causes Data Jockey to be more performant. This result leads us to expect that Data Jockey would improve system utilization despite these initial results.

VII. Related Work

Managing data on HPC systems in the context of scientific workflows gives rise to several data management challenges [23]. Current workflow management systems provide different solutions for managing workflow data. Pegasus [5] uses replica catalogs that map logical file identifiers to their physical locations. However, these catalogs do not support any policy-based data placement and access capabilities as provided by the Data Jockey. Swift [6] is a workflow language that implicitly manages data across multiple tasks and nodes. Swift is capable of utilizing the underlying data transfer protocols for efficient data distribution, but it does not allow users to manage workflow data based on the storage properties of a hierarchical storage system. An alternative way is to use in-situ workflows that minimize the cost of data transfers [24], [25]. But with complex workflows that need to manage data across multiple storage layers, a workflow-aware data management service is required. Different data management strategies are proposed based on the network and storage properties for efficiently transferring data over wide-area networks [26], [27]. Data Jockey is capable of integrating such data management strategies to define policies for efficient data migration across a hierarchical storage system.

VIII. Conclusions

In this work, we have presented Data Jockey to offload the burden of data management from users in HPC environments with a multi-tiered storage architecture. With Data Jockey, we have proposed a new data management scheme called "goal driven data management." By maintaining a view of the current system state and automatically inferring source to destination data movements, Data Jockey allows users to provide only the goal destination of datasets. This scheme eases the complexity of orchestrating data across complex storage hierarchies compared to prior methods where users have to explicitly reason and control the source and destination of datasets. By relieving the user from the programming complexity of tiers, data paths and error handling, our projections show a complexity reduction of 85.7 to 99.9%.

To ascertain whether Data Jockey can be integrated into HPC facilities, we evaluated a prototype implementation in a real HPC environment. Using job and I/O logs from the OLCF, we conducted realistic, synthetic workflows over the course of several days. Aside from mandatory data movements, negligible impact and overhead were observed while Data Jockey was running at scale. Furthermore, by co-scheduling dataset movement and jobs, the combined latency is reduced than when managed independently. Our initial results suggest that Data Jockey would be capable of serving as a center-wide data orchestration service.

As storage tiers continue to deepen and diversify, we believe that systems like Data Jockey will be required to enable scientific progress while not inhibiting system designers from utilizing sophisticated storage architectures.

Acknowledgements

This work was supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE, under the contract No. DE-AC05-00OR22725.

References

[1] J. Lujan *et al.*, "Apex workflows," Technical report, LANL, NERSC, SNL, Tech. Rep., 2015.

[2] S. S. Vazhkudai *et al.*, "The Design, Deployment, and Evaluation of the CORAL Pre-exascale Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018.

[3] "National Enery Research Scientific Computing Center - Cori," http://www.nersc.gov/users/computational-systems/cori/.

[4] I. Altintas, B. Ludaescher, S. Klasky, and M. A. Vouk, "Introduction to scientific workflow management and the kepler system," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, 2006.

[5] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[6] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, 2011.

[7] E. Kakoulli and H. Herodotou, "OctopusFS: A distributed file system with tiered storage management," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, 2017.

[8] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

[9] "Oak Ridge Leadership Computing Facility," https://www.olcf.ornl.gov/.

[10] "Data Management Policy," https://www.olcf.ornl.gov/kb_articles/data-management-policy/.

[11] "iRODS Technical Overview," https://irods.org/uploads/2016/06/technical-overview-2016-web.pdf.

[12] "Robinhood Policy Engine," https://github.com/cea-hpc/robinhood/wiki.

[13] "Kubernetes - Automated container deployment, scaling, and management," http://kubernetes.io.

[14] "Ansible - Automation for everyone," http://www.ansible.com.

[15] "Puppet - Automation for the modern enterprise," http://puppet.com.

[16] *YAML Ain't Markup Language*, http://yaml.org.

[17] "Python," http://www.python.org.

[18] "RabbitMQ," http://www.rabbitmq.com.

[19] "MariaDB Server," https://mariadb.com/products/technology/server.

[20] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.

[21] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*, 2015, pp. 331–342.

[22] "Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x | TOP500 Supercomputer Sites," http://www.top500.org/system/177975.

[23] E. Deelman and A. Chervenak, "Data management challenges of data-intensive scientific workflows," in *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, 2008.

[24] M. Romanus, F. Zhang, T. Jin, Q. Sun, H. Bui, M. Parashar, J. Choi, S. Janhunen, R. Hager, S. Klasky, C.-S. Chang, and I. Rodero, "Persistent data staging services for data intensive in-situ scientific workflows," in *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing DIDC'16*, 2016.

[25] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling in-situ execution of coupled scientific workflow on multi-core platform," in *Proceedings of the 26th International Parallel Distributed Processing Symposium (IPDPS'12)*, 2012.

[26] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, M. Wolf, W. Liao, A. Choudhary *et al.*, "Adaptive io system (adios)," *Cray User's Group*, 2008.

[27] M. Tanaka and O. Tatebe, "Pwrake: A parallel and distributed flexible workflow management tool for wide-area data intensive computing," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, 2010.