CAST: Tiering Storage for Data Analytics in the Cloud

Yue Cheng[⋆], M. Safdar Iqbal[⋆], Aayush Gupta[†], Ali R. Butt[⋆]
[⋆]Virginia Tech; [†]IBM Research – Almaden
{yuec,safdar,butta}@cs.vt.edu, {guptaaa}@us.ibm.com

ABSTRACT

Enterprises are increasingly moving their big data analytics to the cloud with the goal of reducing costs without sacrificing application performance. Cloud service providers offer their tenants a myriad of storage options, which while flexible, makes the choice of storage deployment non trivial. Crafting deployment scenarios to leverage these choices in a cost-effective manner — under the unique pricing models and multi-tenancy dynamics of the cloud environment — presents unique challenges in designing cloud-based data analytics frameworks.

In this paper, we propose CAST, a Cloud Analytics Storage Tiering solution that cloud tenants can use to reduce monetary cost and improve performance of analytics workloads. The approach takes the first step towards providing storage tiering support for data analytics in the cloud. CAST performs offline workload profiling to construct job performance prediction models on different cloud storage services, and combines these models with workload specifications and high-level tenant goals to generate a cost-effective data placement and storage provisioning plan. Furthermore, we build CAST++ to enhance CAST's optimization model by incorporating data reuse patterns and across-jobs interdependencies common in realistic analytics workloads. Tests with production workload traces from Facebook and a 400-core Google Cloud based Hadoop cluster demonstrate that CAST++ achieves 1.21× performance and reduces deployment costs by 51.4% compared to local storage configuration.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*; D.4.8 [**Operating Systems**]: Performance—*Modeling and prediction*

Keywords

Cloud computing; MapReduce; Big data analytics; Storage tiering

Storage type	Capacity (GB/volume)	Throughput (MB/sec)	IOPS (4KB)	Cost ($/month)
`ephSSD`	375	733	100,000	0.218×375
`persSSD`	100	48	3,000	0.17×100
	250	118	7,500	0.17×250
	500	234	15,000	0.17×500
`persHDD`	100	20	150	0.04×100
	250	45	375	0.04×250
	500	97	750	0.04×500
`objStore`	N/A	265	550	0.026/GB

Table 1: Google Cloud storage details. `ephSSD`, `persSSD`, `persHDD` and `objStore` represent VM-local ephemeral SSD, network-attached persistent SSD and HDD, and Google Cloud object storage, respectively. The performance of `persSSD` and `persHDD` scale with volume capacity, whereas `ephSSD` volumes are multiples of 375 GB with a maximum of 4 volumes per VM. Tenants can provision `persSSD` and `persHDD` with a per-volume capacity of up to 10, 240 GB. `objStore` has no storage capacity limit. I/O performance of the local and network-attached block volumes is measured using `fio` and the performance of `objStore` is measured using `gsutil`. All the measured performance numbers match the information provided on [6] (as of Jan. 14, 2015).

1. INTRODUCTION

The cloud computing paradigm provides powerful computation capabilities, high scalability and resource elasticity at reduced operational and administration costs. The use of cloud resources frees tenants from the traditionally cumbersome IT infrastructure planning and maintenance, and allows them to focus on application development and optimal resource deployment. These desirable features coupled with the advances in virtualization infrastructure are driving the adoption of public, private, and hybrid clouds for not only web applications, such as Netflix, Instagram and Airbnb, but also modern big data analytics using parallel programming paradigms such as Hadoop [2] and Dryad [26]. Cloud providers such as Amazon Web Services, Google Cloud, and Microsoft Azure, have started providing data analytics platform as a service [1, 8, 12], which is being adopted widely.

With the improvement in network connectivity and emergence of new data sources such as Internet of Things (IoT) endpoints, mobile platforms, and wearable devices, enterprise-scale data-intensive analytics now involves terabyte- to petabyte-scale data with more data being generated from these sources constantly. Thus, storage allocation and management would play a key role in overall performance improvement and cost reduction for this domain.

While cloud makes data analytics easy to deploy and scale, the vast variety of available storage services with different persistence, performance and capacity characteristics, presents unique challenges for deploying big data analytics in the cloud. For example, Google Cloud Platform provides four different storage options as listed in Table 1. While `ephSSD` offers the highest sequential and random I/O performance, it does not provide data persistence (data stored in `ephSSD` is lost once the associated VMs are terminated). Network-attached persistent block storage services using `persHDD` or `persSSD` as storage media are relatively cheaper than `ephSSD`, but offer significantly lower performance. For instance, a 500 GB `persSSD` volume has about 2× lower throughput and 6× lower IOPS than a 375 GB `ephSSD` volume. Finally, `objStore` is a RESTful object storage service providing the cheapest storage alternative and offering comparable sequential throughput to that of a large `persSSD` volume. Other cloud service providers such as AWS EC2 [5], Microsoft Azure [4], and HP Cloud [10], provide similar storage services with different performance–cost trade-offs.

The heterogeneity in cloud storage services is further complicated by the varying types of jobs within analytics workloads, e.g., iterative applications such as `KMeans` and `Pagerank`, and queries such as `Join` and `Aggregate`. For example, in map-intensive `Grep`, the map phase accounts for the largest part of the execution time (mostly doing I/Os), whereas CPU-intensive `KMeans` spends most of the time performing computation. Furthermore, short-term (within hours) and long-term (daily, weekly or monthly) data reuse across jobs is common in production analytics workloads [18, 15]. As reported in [18], 78% of jobs in Cloudera Hadoop workloads involve data reuse. Another distinguishing feature of analytics workloads is the presence of workflows that represents interdependencies across jobs. For instance, analytics queries are usually converted to a series of batch processing jobs, where the output of one job serves as the input of the next job(s).

The above observations lead to an important question for the cloud tenants *How do I (the tenant) get the most bang-for-the-buck with data analytics storage tiering/data placement in a cloud environment with highly heterogeneous storage resources?* To answer this question, this paper conducts a detailed quantitative analysis with a range of representative analytics jobs in the widely used Google Cloud environment. The experimental findings and observations motivate the design of CAST, which leverages different cloud storage services and heterogeneity within jobs in an analytics workload to perform cost-effective storage capacity allocation and data placement.

CAST does offline profiling of different applications (jobs) within an analytics workload and generates job performance prediction models based on different storage services. It lets tenants specify high-level objectives such as maximizing tenant utility, or minimizing deadline miss rate. CAST then uses a simulated annealing based solver that reconciles these objectives with the performance prediction models, other workload specifications and the different cloud storage service characteristics to generate a data placement and storage provisioning plan. The framework finally deploys the workload in the cloud based on the generated plan. We further enhance our basic tiering design to build CAST++, which incorporates the data reuse and workflow properties of an analytics workload.

Specifically, we make the following contributions in this paper:

1. We employ a detailed experimental study and show, using both qualitative and quantitative approaches, that extant hot/cold data based storage tiering approaches cannot be simply applied to data analytics storage tiering in the cloud.

2. We present a detailed cost-efficiency analysis of analytics workloads and workflows in a real public cloud environment. Our findings indicate the need to carefully evaluate the various storage placement and design choices, which we do, and redesign analytics storage tiering mechanisms that are specialized for the public cloud.

3. Based on the behavior analysis of analytics applications in the cloud, we design CAST, an analytics storage tiering management framework based on simulated annealing algorithm, which searches the analytics workload tiering solution space and *effectively meets* customers' goals. Moreover, CAST's solver *succeeds in discovering non-trivial opportunities* for both performance improvement and cost savings.

4. We extend our basic optimization solver to CAST++ that considers data reuse patterns and job dependencies. CAST++ supports cross-tier workflow optimization using directed acyclic graph (DAG) traversal.

5. We evaluate our tiering solver on a 400-core cloud cluster (Google Cloud) using production workload traces from Facebook. We demonstrate that, compared to a greedy algorithm approach and a series of key storage configurations, CAST++ improves tenant utility by 52.9% – 211.8%, while effectively meeting the workflow deadlines.

2. BACKGROUND AND RELATED WORK

In the following, we provide a brief background of storage tiering, and categorize and compare previous work with our research.

Hot/Cold Data Classification-based Tiering Recent research [28, 34, 43] has focused on improving storage cost and utilization efficiency by placing hot/cold data in different storage tiers. Guerra et. al.[22] builds an SSD-based dynamic tiering system to minimize cost and power consumption, and existing works handle file system and block level I/Os (e.g., 4 – 32 KB) for POSIX-style workloads (e.g., server, database, file systems, etc.). However, the cost model and tiering mechanism used in prior approaches cannot be directly applied to analytics batch processing applications running in a public cloud environment, mainly due to cloud storage and analytics workload heterogeneity. In contrast, our work provides insights into design of a tiered storage management framework for cloud-based data analytics workloads.

Fine-Grained Tiering for Analytics Storage tiering has been studied in the context of data-intensive analytics batch applications. Recent analysis [23] demonstrates that adding a flash tier for serving reads is beneficial for HDFS-based HBase workloads with random I/Os. As opposed to HBase I/O characteristics, typical MapReduce-like batch jobs issues large, sequential I/Os [40] and run in multiple stages (map, shuffle, reduce). Hence, lessons learned from HBase tiering are not directly applicable to such analytics workloads. hatS [31] and open source Hadoop community [9] have taken the first steps towards integrating heterogeneous storage devices in HDFS for local clusters. However, the absence of task-level tier-aware scheduling mechanisms implies that these HDFS block granularity tiering approaches cannot avoid stragglers within a job, thus achieving limited performance gains if any. PACMan [15] solves this slow-tier straggler problem by using a memory caching policy for small jobs whose footprint can fit in the memory of the cluster. Such caching approaches are complementary to CAST as it provides a coarse-grained, static data placement solution for a complete analytics workload in different cloud storage services.

Cloud Resource Provisioning Considerable prior work has examined ways to automate resource configuration and provisioning process in the cloud. Frugal Cloud File System (FCFS) [39] is a cost-effective cloud-based file storage that spans multiple cloud storage services. In contrast to POSIX file system workloads, modern analytics jobs (focus of our study) running on parallel programming frameworks like Hadoop demonstrate very different access characteristics and data dependencies (described in Section 3); requiring a rethink of how storage tiering is done to benefit these workloads. Other works such as Bazaar [27] and Conductor [44], focus on automating cloud resource deployment to meet cloud tenants' requirements while reducing deployment cost. Our work takes a thematically similar view — exploring the trade-offs of cloud services — but with a different scope that targets data analytics workloads and leverages their unique characteristics to provide storage tiering. Several systems [14, 37] are specifically designed to tackle flash storage allocation inefficiency in virtualization platforms. In contrast, we explore the inherent performance and cost trade-off of different storage services in public cloud environments.

Analytics Workflow Optimization A large body of research [45, 35, 21, 36, 33] focuses on Hadoop workflow optimizations by integrating workflow-aware scheduler into Hadoop or interfacing Hadoop with a standalone workflow scheduler. Our workflow enhancement is orthogonal and complements these works as well — CAST++ exploits cloud storage heterogeneity and performance scaling property, and uses opportunities for efficient data placement across different cloud storage services to improve workflow execution. Workflow-aware job schedulers can leverage the data placement strategy of CAST++ to further improve analytics workload performance.

3. A CASE FOR CLOUD STORAGE TIERING

In this section, we first establish the need for cloud storage tiering for data analytics workloads. To this end, we characterize the properties of applications that form a typical analytics workload and demonstrate the impact of these properties on the choice of cloud storage services. We then argue that extant tiering techniques, such as hot/cold data based segregation and fine-grained partitioning within a single job, are not adequate; *rather a course-grained, job-level storage service tiering is needed for cloud-based data analytics.*

App.	**I/O-intensive**			**CPU-intensive**
	Map	Shuffle	Reduce	
Sort	✗	✓	✗	✗
Join	✗	✓	✓	✗
Grep	✓	✗	✗	✗
KMeans	✗	✗	✗	✓

Table 2: Characteristics of studied applications.

3.1 Characterization of Data Analytics Workloads

We characterize the analytics workloads along two dimensions. First, we study the behavior of individual applications within a large workload when executed on parallel programming paradigms such as MapReduce — demonstrating the benefits of different storage services for various applications. Second, we consider the role of cross-job relationships (an analytics workload comprises multiple jobs each executing an application) and show how these interactions affect the choice of efficient data placement decisions for the same applications.

3.1.1 Experimental Study Setup

We select four representative analytics applications that are typical components of real-world analytics workloads [18, 46] and exhibit diversified I/O and computation characteristics, as listed in Table 2. `Sort`, `Join` and `Grep` are I/O-intensive applications. The execution time of `Sort` is dominated by the shuffle phase I/O, transferring data between mappers and reducers. In contrast, `Grep` spends most of its runtime in the map phase I/O, reading the input and finding records that match given patterns. `Join` represents an analytics query that combines rows from multiple tables and performs the join operation during the reduce phase, and thus is reduce intensive. `KMeans` is an iterative machine learning clustering application that spends most of its time in the compute phases of map and reduce iterations, which makes it CPU-intensive.

The experiments are performed in Google Cloud using a `n1-standard-16` VM (16 vCPUs, 60 GB memory) with the master node on a `n1-standard-4` VM (4 vCPUs, 15 GB memory). Intermediate data is stored on the same storage service as the original data, except for `objStore`, where we use `persSSD` for intermediate storage. Unless otherwise stated, all experiments in this section are conducted using the same compute resources but with different storage configurations as stated.

3.1.2 Analysis: Application Granularity

Figure 1 depicts both the execution time of the studied applications and tenant utility for different choices of storage services. We define *tenant utility* (or simply "utility," used interchangeably) to be $\frac{1/execution\ time}{cost\ in\ dollars}$. This utility metric is based on the tenants' economic constraints when deploying general workloads in the cloud. Figure 1 (a) shows that `ephSSD` serves as the best tier for both execution time and utility for `Sort` even after accounting for the data transfer cost for both upload and download from `objStore`. This is because there is no data reduction in the map phase and the entire input size is written to intermediate files residing on `ephSSD` that has about 2× higher sequential bandwidth than `persSSD`. Thus, we get better utility from `ephSSD` than

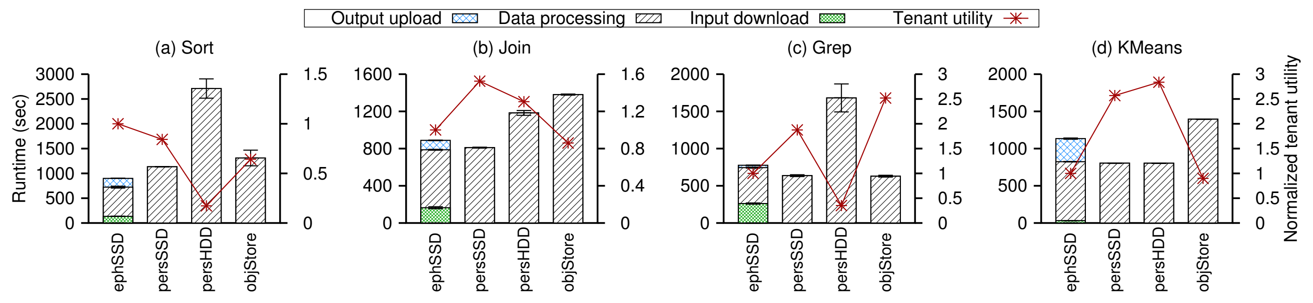


Figure 1: Application performance and achieved tenant utility on different cloud storage tiers. `ephSSD` does not offer persistence, so we account for data transfer time (i.e., input download time from `objStore` and output upload time to `objStore`) and break down the `ephSSD` runtime into three parts. Applications that run on `ephSSD` incur storage cost that includes both the cost of `ephSSD` and `objStore`. Applications that run on `objStore` require either a local ephemeral disk or network-attached persistent disk for storing intermediate data. For this set of experiments, we used a 100 GB `persSSD` as intermediate data store. Each bar represents the average from three runs. Tenant utility is normalized to that of `ephSSD`.

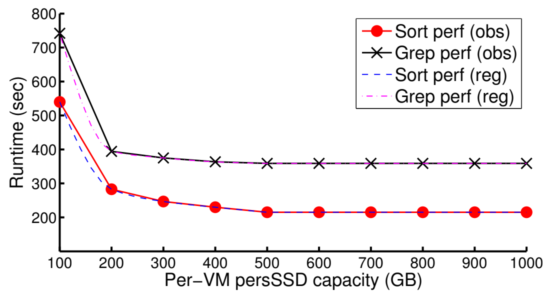


Figure 2: Impact of scaling `persSSD` volume capacity for `Sort` and `Grep`. The regression model is used in CAST's tiering approach and is described in detail in §4. These tests are conducted on a 10-VM `n1-standard-16` cluster.

`persSSD`, albeit at a slightly higher cost. On the other hand, Figure 1 (b) shows that, `Join` works best with `persSSD`, while it achieves the worst utility on `objStore`. This is due to high overheads of setting up connections to request data transfers using the Google Cloud Storage Connector (GCS connector) for Hadoop APIs [7] for the many small files generated by the involved reduce tasks. `Grep`'s map-intensive feature dictates that its performance solely depends on sequential I/O throughput of the storage during the map phase. Thus, in Figure 1 (c) we observe that both `persSSD` and `objStore` provide similar performance (both have similar sequential bandwidth as seen in Table 1) but the lower cost of `objStore` results in about 34.3% higher utility than `persSSD`. Similarly, for the CPU-bound `KMeans`, while `persSSD` and `persHDD` provide similar performance, the lower cost of `persHDD` yields much better utility as shown in Figure 1 (d).

Performance Scaling In Google Cloud, performance of network-attached block storage depends on the size of the volume, as shown in Table 1. Other clouds such as Amazon AWS offer different behavior but typically the block storage performance in these clouds can be scaled by creating logical volumes by striping (RAID-0) across multiple network-attached block volumes. In Figure 2, we study the impact of this capacity scaling on the execution time of two I/O-intensive applications, `Sort` and `Grep` (we also observe similar patterns for other I/O-intensive applications). For a network-attached `persSSD` volume, the dataset size of `Sort` is 100 GB and that of `Grep` is 300 GB. We observe that as the volume capacity increases from 100 GB to 200 GB, the run time of both `Sort` and `Grep` is reduced by 51.6% and 60.2%, respectively. Any further increase in capacity offers marginal benefits. This happens because in both these applications the I/O bandwidth bottleneck is alleviated when the capacity is increased to 200 GB. Beyond that, the execution time is dependent on other parts of the MapReduce framework. These observations imply that it is possible to achieve desired application performance in the cloud without resorting to unnecessarily over-provisioning of the storage and thus within acceptable cost.

Key Insights From our experiments, we infer the following. (i) There is no one storage service that provides the best raw performance as well as utility for different data analytics applications. (ii) For some applications, slower storage services, such as `persHDD`, may provide better utility and comparable performance to other costlier alternatives. (iii) Elasticity and scalability of cloud storage services should be leveraged through careful over-provisioning of capacity to reduce performance bottlenecks in I/O intensive analytics applications.

3.1.3 *Analysis: Workload Granularity*

We next study the impact of cross-job interactions within an analytics workload. While individual job-level optimization and tiering has been the major focus of a number of recent works [31, 32, 44, 25, 24, 17], we argue that this is not sufficient for data placement in the cloud for analytics workloads. To this end, we analyze two typical workload characteristics that have been reported in production workloads [18, 15, 38, 23, 19], namely data reuse across jobs, and dependency between jobs, i.e., workflows, within a workload.

Data Reuse across Jobs As reported in the analysis of production workloads from Facebook and Microsoft Bing [18, 15], both small and large jobs exhibit data re-access patterns both in the short term, i.e., input data shared by multiple jobs and reused for a few hours before becoming cold, as well as in the long term, i.e., input data reused for longer periods such as days or weeks before turning cold. Henceforth, we refer to the former as reuse-lifetime (short) and the later as reuse-lifetime (long).

To better understand how data reuse affects data placement choices, we evaluate the tenant utility of each application under different reuse patterns. Figure 3 shows that the choice of storage service changes based on data reuse patterns for different applications. Note that in both reuse

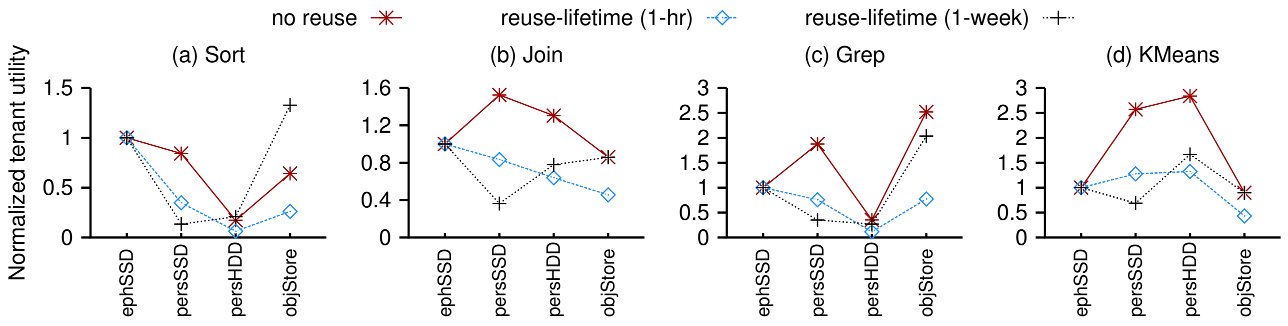


Figure 3: Tenant utility under different data reuse patterns. `reuse-lifetime (1 hr)` represents re-accesses of the same data over a period of 1 hour and `reuse-lifetime (1 week)` represents data re-accesses over 1 week. Tenant utility is normalized to that of `ephSSD`.

cases we perform the same number of re-accesses (i.e., 7) over the specified time period. For instance, in `reuse-lifetime (1 week)`, data is accessed once per day, i.e., 7 accesses in a week. Similarly, data is accessed once every 8 minutes in `reuse-lifetime (1 hr)` [1]. For `ephSSD`, the input download overheads can be amortized by keeping the data in the ephemeral SSD, since the same data will be re-accessed in a very short period of time. This allows `ephSSD` to provide the highest utility for `Join` (Figure 3 (b)) and `Grep` (Figure 3 (c)) for `reuse-lifetime (1 hr)`. However, if the data is re-accessed only once per day (`reuse-lifetime (1 week)`), the cost of `ephSSD` far outweighs the benefits of avoiding input downloads. Thus, for `Sort` (Figure 3 (a)), `objStore` becomes the storage service of choice for `reuse-lifetime (1 week)`. For similar cost reasons, `persSSD`, which demonstrates the highest utility for individual applications (Figure 1 (b)), becomes the worst choice when long term re-accesses are considered. Furthermore, as expected, the behavior of CPU-intensive `KMeans` (Figure 3 (d)) remains the same across reuse patterns with highest utility achieved with `persHDD`, and the storage costs do not play a major role in its performance.

Workflows in an Analytics Workload An analytics workflow consists of a series of jobs with inter-dependencies. For our analysis, we consider the workflows where the output of one job acts as a part of an input of another job. Thus, a workflow can be abstracted as Directed Acyclic Graph (DAGs) of actions [3]. Prior research [33, 35] has shown that not only overall completion times of individual jobs in a workflow an important consideration, but meeting completion time deadlines of workflows is also critical for satisfying Service-Level Objectives (SLOs) of cloud tenants.

Consider the following example to illustrate and support the above use case. Figure 4(a)[2] lists four possible tiering plans for a four-job workflow. The workflow consists of four jobs and represents a typical search engine log analysis. Figure 4(a) (i) and Figure 4(a) (ii) depict cases where a single storage service, `objStore` and `persSSD`, respectively, is used for the entire workflow. As shown in Figure 4(b), the complex nature of the workflow not only makes these two data placement strategies perform poorly (missing a hypothetical deadline of 8,000 seconds) but also results in high costs compared to the other two hybrid storage plans. On the other hand, both the hybrid storage services meet the deadline. Here, the output of one job is pipelined to another storage service where it acts as an input for the subsequent job in the workflow. If the tenant's goal is to pick a strategy that provides the lowest execution time, then the combination `objStore+ephSSD` shown in Figure 4(b) provides the best result amongst the studied plans. However, if the tenant wants to choose a layout that satisfies the dual criteria of meeting the deadline and providing the lowest cost (among the considered plans), then the combination `objStore+ephSSD+persSSD` — that reduces the cost by 7% compared to the other tiering plan — may be a better fit.

Key Insights From this set of experiments, we infer the following. (i) Not only do data analytics workloads require use of different storage services for different applications, the data placement choices also change when data reuse effects are considered. (ii) Complex inter-job requirements in workflows necessitate thinking about use of multiple storage services, where outputs of jobs from a particular storage service may be pipelined to different storage tiers that act as inputs for the next jobs. (iii) Use of multiple criteria by the tenant, such as workflow deadlines and monetary costs, adds more dimensions to a data placement planner and requires careful thinking about tiering strategy.

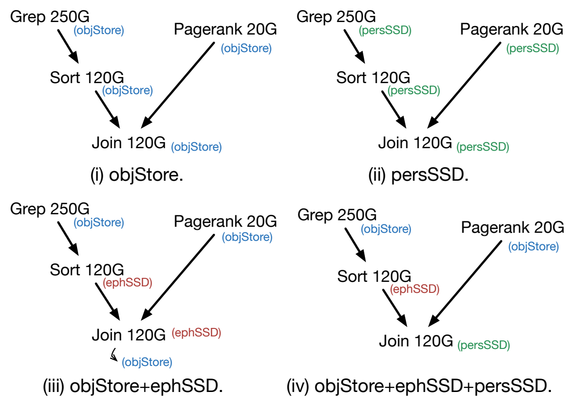
3.2 Shortcomings of Traditional Storage Tiering Strategies

In this following, we argue that traditional approaches to storage tiering are not adequate for being used for analytics workloads in the cloud.

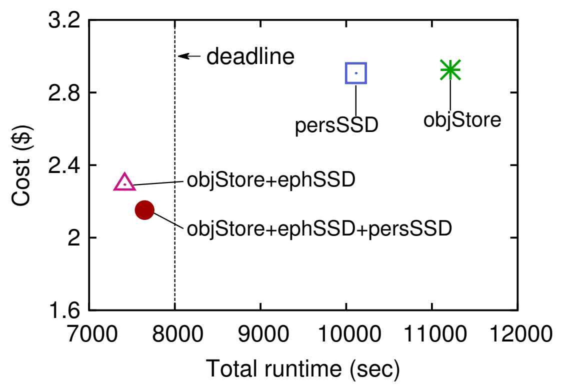
Heat-based Tiering A straw man tiering approach that considers the monetary cost of different cloud storage mediums is to place hot data on `ephSSD`; semi-hot data on either `persSSD` or `persHDD`; and cold data on the cheapest `objStore`. Heat metrics can include different combinations of access frequencies, recency, etc. But the performance–cost model for cloud storage for analytics workloads is more complicated due to the following reasons. (1) The most expensive cloud storage tier (`ephSSD`) may not be the best tier for hot data, since the ephemeral SSD tier typically provides no persistence guarantee — the VMs have to persist for ensuring that all the data on ephemeral disks stays available, potentially increasing monetary costs. `ephSSD` volumes are fixed in size (Table 1) and only 4 volumes can be attached to a VM. Such constraints can lead to both under-provisioning (requiring

[1]While re-access frequency can vary for different reuse-lifetimes, we selected these two cases to highlight the changes in data placement options for the same applications due to different data reuse patterns.

[2]We do not show utility results of `Pagerank` because it exhibits the same behavior as `KMeans` in §3.1.2.



(a) Four possible workflow tiering plans.



(b) Workflow tiering performance-cost trade-offs.

Figure 4: Possible tiering plans for a simple 4-job workflow. Storage medium in parentheses indicates where the input of the job is stored. For instance, `Pagerank 20G (objStore) -> Join 120G (persSSD)` means that the input and intermediate data of job `Pagerank` is stored on `objStore`, and the output is placed on `persSSD` and then consumed by the job `Join` as input. The output of `Pagerank` is 386 MB and consists of `pageIDs`, and hence is not shown being added to the input of `Join`.

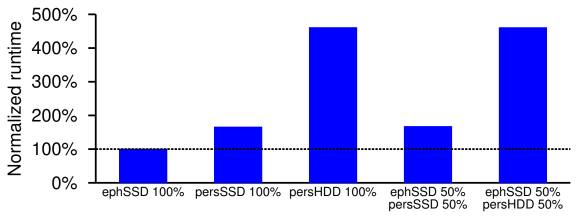
more VMs to be started) and over-provisioning (wasting capacity for small datasets), in turn reducing utility. (2) Furthermore, analytics applications may derive better utility from cheaper tiers than their more expensive counterparts.

Fine-Grained Tiering Recently, hatS [31] looked at a tiered HDFS implementation that utilizes both HDDs and SSDs to improve single job performance. Such approaches focus on fine-grained tiering within a single job. While this can be useful for on-premise clusters where storage resources are relatively fixed and capacity of faster storage is limited [23], it provides few benefits for cloud-based analytics workloads where resources can be elastically provisioned. Furthermore, maintaining heat metrics at a fine-grained block or file level may be spatially prohibitive (DRAM requirements) for a big data analytics workload with growing datasets.

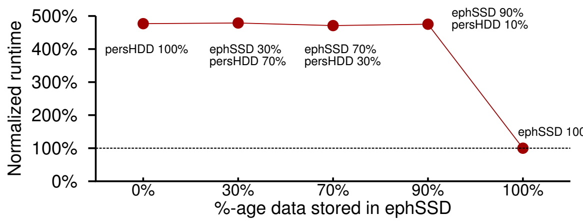
We contend that instead of looking at block or file-level partitioning, a more coarse-grained approach that performs job-level tiering, is needed for cloud-based analytics. To illustrate this, in Figure 5 we measure the performance of `Grep` under various placement configurations (using default Hadoop task scheduler and data placement policy) for a 6 GB input dataset requiring 24 map tasks scheduled as a single wave. As shown in Figure 5(a), partitioning data across a faster `ephSSD` and slower `persSSD` tier does not improve performance. The tasks on slower storage media dominate the execution time. We further vary the partitioning by increasing the fraction of input data on faster `ephSSD` (Figure 5(b)). We observe that even if 90% of the data is on the faster tier, the performance of the application does not improve, highlighting the need for job-level data partitioning. Such an "all-or-nothing" [15] data placement policy, i.e., placing the whole input of one job in one tier, is likely to yield good performance. This policy is not only simple to realize, but also maps well to both the characteristics of analytics workloads and elasticity of cloud storage services.

4. CAST FRAMEWORK

We build Cast, an analytics storage tiering framework that exploits heterogeneity of both the cloud storage and analytics workloads to satisfy the various needs of cloud



(a) `ephSSD+persSSD`, `ephSSD+persHDD` are hybrid storage configurations.



(b) Fine-grained data partitioning within a job. Data is partitioned across faster `ephSSD` and slower `persHDD`.

Figure 5: Normalized runtime of `Grep` under different HDFS configurations. All runtime numbers are normalized to `ephSSD 100%` performance.

tenants. Furthermore, Cast++, an enhancement to Cast, provides data pattern reuse and workflow awareness based on the underlying analytics framework. Figure 6 shows the high-level overview of Cast operations and involves the following components. (1) The analytics *job performance estimator* module evaluates jobs execution time on different storage services using workload specifications provided by tenants. These specifications include a list of jobs, the application profiles, and the input data sizes for the jobs. The estimator combines this with compute platform information to estimate application run times on different storage services. (2) The *tiering solver* module uses the job execution estimates from the *job performance estimator* to generate a tiering plan that spans all storage tiers on the specific cloud provider available to the tenant. The objective of the solver

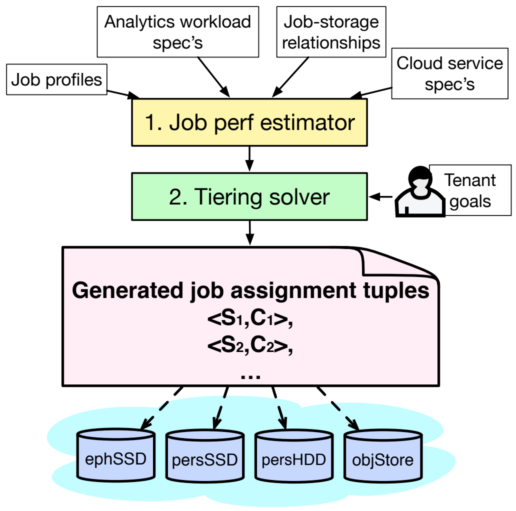


Figure 6: Overview of CAST tiering framework.

is to satisfy the high-level tenants' goals such as achieving high utility or reducing deadline miss rates.

4.1 Estimating Analytics Job Performance

The well-defined execution phases of the MapReduce parallel programming paradigm [20, 27] implies that the runtime characteristics of analytics jobs can be predicted with high accuracy. Moreover, extensive recent research has focused on data analytics performance prediction [42, 25, 24, 13, 41, 27]. We leverage and adapt MRCute [27] model in CAST to predict job execution time, due to its ease-of-use, availability, and applicability to our problem domain.

Equation 1 defines our performance prediction model. It consists of three sub-models — one each for the map, shuffle, and reduce phases — where each phase execution time is modeled as $\#waves \times runtime\ per\ wave$. A *wave* represents the number of tasks that can be scheduled in parallel based on the number of available slots. CAST places all the data of a job on a single storage service with predictable performance, and tasks (within a job) work on equi-sized data chunks. The estimator also models wave pipelining effects — in a typical MapReduce execution flow, the three phases in the same wave are essentially serialized, but different waves can be overlapped. Thus, the prediction model does not sacrifice estimation accuracy. The model simplifies the predictor implementation, which is another advantage of performing coarse-grained, job-level data partitioning.

$$EST(\hat{\mathcal{R}},\hat{\mathcal{M}}(s_i,\hat{\mathcal{L}}_i)) = \overbrace{\underbrace{\left\lceil \frac{m}{n_{vm}\cdot m_c}\right\rceil}_{\#\ \text{waves}} \cdot \underbrace{\left(\frac{input_i/m}{bw^{s_i}_{map}}\right)}_{\text{Runtime per wave}}}^{\text{map phase}} + \underbrace{\left\lceil \frac{r}{n_{vm}\cdot r_c}\right\rceil \cdot \left(\frac{inter_i/r}{bw^{s_i}_{shuffle}}\right)}_{\text{shuffle phase}} + \underbrace{\left\lceil \frac{r}{n_{vm}\cdot r_c}\right\rceil \cdot \left(\frac{output_i/r}{bw^{s_i}_{reduce}}\right)}_{\text{reduce phase}} . \quad (1)$$

The estimator $EST(.)$ predicts job performance using the information about (i) job configuration: number of map/reduce tasks, job sizes in different phases; (ii) compute configuration: number of VMs, available slots per VM; and (iii) storage services: bandwidth of tasks on a particular storage service. Table 3 lists the notations used in the model.

	Notation	Description
$\hat{\mathcal{R}}$	n_{vm}	number of VMs in the cluster
	m_c	number of map slots in one node
	r_c	number of reduce slots in one node
$\hat{\mathcal{M}}$	bw^f_{map}	bandwidth of a single map task on tier f
	$bw^f_{shuffle}$	bandwidth of a single shuffle task on tier f
	bw^f_{reduce}	bandwidth of a single reduce task on tier f
$\hat{\mathcal{L}}_i$	$input_i$	input data size of job i
	$inter_i$	intermediate data size of job i
	$output_i$	output data size of job i
	m	number of map tasks of job i
	r	number of reduce tasks of job i
solver	$capacity$	total capacities of different storage mediums
	$price_{vm}$	VM price (\$/min)
	$price_{store}$	storage price (\$/GB/hr)
	J	set of all analytics jobs in a workload
	J_w	set of all analytics jobs in a workflow w
	F	set of all storage services in the cloud
	D	set of all jobs that share the same data
	$\hat{P}$	tiering solution
decision vars	s_i	storage service used by job i
	c_i	storage capacity provisioned for job i

Table 3: Notations used in the analytics jobs performance prediction model and CAST tiering solver.

4.2 Basic Tiering Solver

The goal of the basic CAST tiering solver is to provide near-optimal specification that can help guide tenant's decisions about data partitioning on the available storage services for their analytics workload(s). The solver uses a simulated annealing algorithm [29] to systematically search through the solution space and find a desirable tiering plan, given the workload specification, analytics models, and tenants' goals.

4.2.1 CAST Solver: Modeling

The data placement and storage provisioning problem is modeled as a non-linear optimization problem that maximizes the tenant utility (U) defined in Equation 2:

$$max\ \ U = \frac{{}^{1}/_{T}}{(\$_{vm} + \$_{store})} \ , \quad (2)$$

$$s.t.\ \ c_i \geq (input_i + inter_i + output_i)\ (\forall i \in J) \ , \quad (3)$$

$$T = \sum_{i=1}^{J} REG\big(s_i, capacity[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}}_i\big)\ , where\ s_i \in F \ , \quad (4)$$

$$\$_{vm} = n_{vm} \cdot (price_{vm} \cdot T) \ , \quad (5)$$

$$\$_{store} = \sum_{f=1}^{F} \Big(capacity[f] \cdot \big(price_{store}[f] \cdot \left\lceil {}^{T}/_{60} \right\rceil\big)\Big) \quad (6)$$

$$where\ \forall f \in F\ :\Big\{\forall i \in J,\ s.t.\ s_i \equiv f : capacity[f] = \sum c_i\Big\} .$$

The performance is modeled as the *reciprocal* of the estimated completion time in minutes (${}^{1}/_{T}$) and the costs include both the VM and storage costs. The VM cost[3] is defined by Equation 5 and depends on the total completion time of the workload. The cost of each storage service is determined by the workload completion time (storage cost is

[3]We only consider a single VM type since we focus on storage tiering. Extending the model to incorporate heterogeneous VM types is part of our future work.

charged on a hourly basis) and capacity provisioned for that service. The overall storage cost is obtained by aggregating the individual costs of each service (Equation 6).

Equation 3 defines the *capacity constraint*, which ensures that the storage capacity (c_i) provisioned for a job is sufficient to meet its requirements for all the phases (map, shuffle, reduce). We also consider intermediate data when determining aggregated capacity. For jobs, e.g., Sort, which have a selectivity factor of one, the intermediate data is of the same size as the input data. Others, such as inverted indexing, would require a large capacity for storing intermediate data as significant larger shuffle data is generated during the map phase [16]. The generic Equation 3 accounts for all such scenarios and guarantees that the workload will not fail. Given a specific tiering solution, the estimated total completion time of the workload is defined by Equation 4. Since job performance in the cloud scales with capacity of some services, we use a regression model, $REG(s_i,\ .)$, to estimate the execution time. In every iteration of the solver, the regression function uses the storage service (s_i) assigned to a job in that iteration, the total provisioned capacity of that service for the entire workload, cluster information such as number of VMs and the estimated runtime based on Equation 1 as parameters. After carefully considering multiple regression models, we find that a third degree polynomial-based cubic Hermite spline [30] is a good fit for the applications and storage services considered in the paper. While we do not delve into details about the model, we show the accuracy of the splines in Figure 2. We also evaluate the accuracy of this regression model using a small workload in §5.1.

4.2.2 CAST Solver: Algorithms

Algorithm 1: Greedy static tiering algorithm.

```
Input: Job information matrix: $\hat{\mathcal{L}}$,
Output: Tiering plan $\hat{P}_{greedy}$
begin
    $\hat{P}_{greedy} \leftarrow \{\}$
    foreach $job\ j\ in\ \hat{\mathcal{L}}$ do
        $f_{best} \leftarrow f_1$        // $f_1$ represents the first of the
                                        // available storage services in $F$
        foreach $storage\ service\ f_{curr}\ in\ F$ do
            if $Utility(j, f_{curr}) > Utility(j, f_{best})$ then
                $f_{best} \leftarrow f_{curr}$
        $\hat{P}_{greedy} \leftarrow \hat{P}_{greedy} \cup \{\langle j, f_{best}\rangle\}$
    return $\hat{P}_{greedy}$
```

Greedy Algorithm We first attempt to perform data partitioning and placement using a simple greedy algorithm (Algorithm 1). The algorithm takes the job information matrix J as the input and generates a tiering plan as follows. For each job in the workload, the utility (calculated using function $Utility(.)$) is computed using Equation 1 and Equation 2 on each storage service. The tier that offers the highest utility is assigned to the job. As astute readers will observe, while this algorithm is straightforward to reason about and implement, it does not consider the impact of placement on other jobs in the workload. Furthermore, as we greedily make placement decisions on a per-job basis, the total provisioned capacity on a tier increases. Recall that the performance of some storage services scales with capacity. Thus, the tiering decisions for some jobs (for which placement has already been done) may no longer provide maximum utility. We evaluate the impact of these localized greedy decisions in §5.1.

Algorithm 2: Simulated Annealing Algorithm.

```
Input: Job information matrix: $\hat{\mathcal{L}}$ ,
       Analytics job model matrix: $\hat{\mathcal{M}}$ ,
       Runtime configuration: $\hat{\mathcal{R}}$ ,
       Initial solution: $\hat{P}_{init}$ .
Output: Tiering plan $\hat{P}_{best}$
begin
    $\hat{P}_{best} \leftarrow \{\}$
    $\hat{P}_{curr} \leftarrow \hat{P}_{init}$
    $exit \leftarrow False$
    $iter \leftarrow 1$
    $temp_{curr} \leftarrow temp_{init}$
    $U_{curr} \leftarrow Utility(\hat{\mathcal{M}}, \hat{\mathcal{L}}, \hat{P}_{init})$
    while $not\ exit$ do
        $temp_{curr} \leftarrow Cooling(temp_{curr})$
        for $next\ \hat{P}_{neighbor}\ in\ AllNeighbors(\hat{\mathcal{L}}, \hat{P}_{curr})$ do
            if $iter > iter_{max}$ then
                $exit \leftarrow True$
                break
            $U_{neighbor} \leftarrow Utility(\hat{\mathcal{M}}, \hat{\mathcal{L}}, \hat{P}_{neighbor})$
            $\hat{P}_{best} \leftarrow UpdateBest(\hat{P}_{neighbor}, \hat{P}_{best})$
            $iter{+}{+}$
            if $Accept(temp_{curr},\ U_{curr},\ U_{neighbor})$ then
                $\hat{P}_{curr} \leftarrow \hat{P}_{neighbor}$
                $U_{curr} \leftarrow U_{neighbor}$
                break
    return $\hat{P}_{best}$
```

Simulated Annealing-based Algorithm In order to overcome the limitations of the Greedy approach, we devise a simulated annealing [29] based algorithm. The algorithm (Algorithm 2) takes as input workload information ($\hat{\mathcal{L}}$), compute cluster configuration ($\hat{\mathcal{R}}$), and information about performance of analytics applications on different storage services ($\hat{\mathcal{M}}$) as defined in Table 3. Furthermore, the algorithm uses $\hat{P}_{init}$ as the initial tiering solution that is used to specify preferred regions in the search space. For example, the results from the greedy algorithm or the characteristics of analytics applications described in Table 2 can be used to devise an initial placement.

The main goal of our algorithm is to find a near-optimal tiering plan for a given workload. In each iteration, we pick a randomly selected neighbor of the current solution ($AllNeighbors(.)$). If the selected neighbor yields better utility, it becomes the current best solution. Otherwise, in the function $Accept(.)$, we decide whether to move the search space towards the neighbor ($\hat{P}_{neighbor}$) or keep it around the current solution ($\hat{P}_{curr}$). This is achieved by considering the difference between the utility of the current (U_{curr}) and neighbor solutions ($U_{neighbor}$) and comparing it with a distance parameter, represented by $temp_{curr}$. In each iteration, the distance parameter is adjusted (decreased) by a $Cooling(.)$ function. This helps in making the search narrower as iterations increase; reducing the probability of missing the maximum utility in the neighborhood of the search space.

4.3 Enhancements: CAST++

While the basic tiering solver improves tenant utility for general workloads, it is not able to leverage certain properties of analytics workloads. To this end, we design CAST++, which enhances CAST by incorporating data reuse patterns and workflow awareness.

Enhancement 1: Data Reuse Pattern Awareness To incorporate data reuse patterns across jobs, CAST++ ensures that all jobs that share the same input dataset have the same storage service allocated to them. This is captured by Constraint 7 where D represents the set consisting of jobs sharing the input (partially or fully).

$$s_i \equiv s_l \; (\forall i \in D, ß \neq l, \in D) \tag{7}$$
$$where\ D = \{all\ jobs\ which\ share\ input\}$$

Thus, even though individual jobs may have different storage tier preferences, Cast++ takes a global view with the goal to maximize overall tenant utility.

Enhancement 2: Workflow Awareness Prior research has shown that analytics workflows are usually associated with a tenant-defined deadline [33, 21]. For workloads with a mix of independent and inter-dependent jobs, the basic dependency-oblivious Cast may either increase the deadline miss rate or unnecessarily increase the costs, as we show in §3.1.3. Hence, it is crucial for Cast++ to handle workflows differently. To this end, we enhance the basic solver to consider the objective of minimizing the total monetary cost (Equation 8) and introduce a constraint to enforce that the total estimated execution time meets the predefined deadline (Equation 9). This is done for optimizing each workflow separately. Each workflow is represented as a Directed Acyclic Graph (DAG) where each vertex is a job and a directed edge represents a flow from one job to another (refer to Figure 4(a)).

$$min\ \$_{total} = \$_{vm} + \$_{store}\ , \tag{8}$$

$$s.t.\ \sum_{i=1}^{J_w} REG\Big(s_i, s_{i+1}, capacity[s_i], \hat{\mathcal{R}}, \hat{\mathcal{L}}\Big) \leq deadline\ , \tag{9}$$

$$c_i \geq \sum_{i=1}^{J_w} \big((s_{i-1} \neq s_i) \cdot input_i + inter_i \tag{10}$$
$$+ (s_{i+1} \equiv s_i) \cdot output_i\big)\ , where\ s_0 = \phi\ .$$

Furthermore, Equation 10 restricts the capacity constraint in Equation 3 by incorporating inter-job dependencies. The updated approach only allocates capacity if the storage tier for the output of a job at the previous level is not the same as input storage tier of a job at the next level in the DAG. To realize this approach, we enhance Algorithm 2 by replacing the next neighbor search ($AllNeighbors(.)$) with a depth-first traversal in the workflow DAG. This allows us to reduce the deadline miss rate.

5. EVALUATION

In this section, we present the evaluation of Cast and Cast++ using a 400-core Hadoop cluster on Google Cloud. Each slave node in our testbed runs on a 16 vCPU `n1-standard-16` VM as specified in §3. We first evaluate the effectiveness of our approach in achieving the best tenant utility for a 100-job analytics workload with no job dependencies. Then, we examine the efficacy of Cast++ in meeting user-specified deadlines.

5.1 Tenant Utility Improvement

5.1.1 Methodology

We compare Cast against six storage configurations: four without tiering and two that employ greedy algorithm based static tiering. We generate a representative 100-job workload by sampling the input sizes from the distribution observed in production traces from a 3,000-machine Hadoop deployment at Facebook [18]. We quantize the job sizes into 7 bins as listed in Table 4, to enable us to compare the dataset size distribution across different bins. The largest job in the Facebook traces has 158,499 map tasks. Thus, we choose 3,000 for the highest bin in our workload to ensure that our workload demands a reasonable load but is also manageable for our 400-core cluster. More than 99% of the total data in the cluster is touched by the large jobs that belong to bin 5, 6 and 7, which incur most of the storage cost. The aggregated data size for small jobs (with number of map tasks in the range 1–10) is only 0.1% of the total data size. The runtime for small jobs is not sensitive to the choice of storage tier. Therefore, we focus on the large jobs, which have enough number of mappers and reducers to fully utilize the cluster compute capacity during execution. Since there is a moderate amount of data reuse throughout the Facebook traces, we also incorporate this into our workload by having 15% of the jobs share the same input data. We assign the four job types listed in Table 2 to this workload in a round-robin fashion to incorporate the different computation and I/O characteristics.

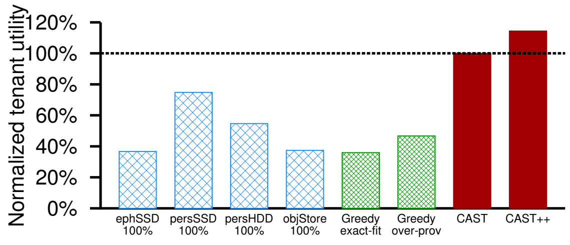
Bin	# Maps at Facebook	% Jobs at Facebook	% Data sizes at Facebook	# Maps in workload	# Jobs in workload
1				1	35
2	1—10	73%	0.1%	5	22
3				10	16
4	11—50	13%	0.9%	50	13
5	51—500	7%	4.5%	500	7
6	501—3000	4%	16.5%	1,500	4
7	> 3000	3%	78.1%	3,000	3

Table 4: Distribution of job sizes in Facebook traces and our synthesized workload.

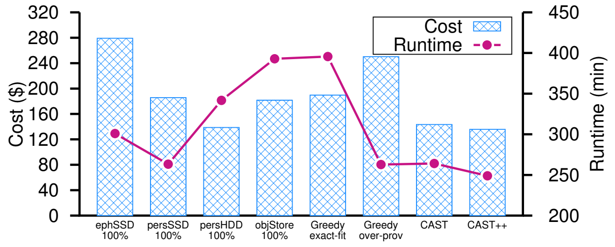
5.1.2 Effectiveness for General Workload

Figure 7 shows the results for tenant utility, performance, cost and storage capacity distribution across four different storage services. We observe in Figure 7(a) that Cast improves the tenant utility by 33.7% – 178% compared to the configurations with no explicit tiering, i.e., `ephSSD 100%`, `persSSD 100%`, `persHDD 100%` and `objStore 100%`. The best combination under Cast consists of 33% `ephSSD`, 31% `persSSD`, 16% `persHDD` and 20% `objStore`, as shown in Figure 7(c). `persSSD` achieves the highest tenant utility among the four non-tiered configurations, because `persSSD` is relatively fast and persistent. Though `ephSSD` provides the best I/O performance, it is not cost-efficient, since it uses the most expensive storage and requires `objStore` to serve as the backing store to provide data persistence, which incurs additional storage cost and also imposes data transfer overhead. This is why `ephSSD 100%` results in 14.3% longer runtime (300 minutes) compared to that under `persSSD 100%` (263 minutes) as shown in Figure 7(b).

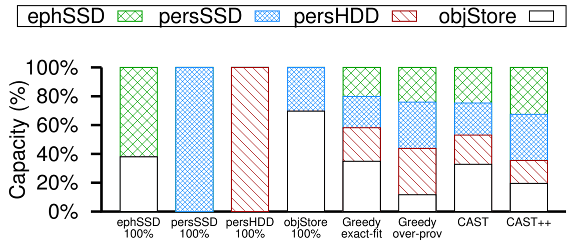
The greedy algorithm cannot reach a global optimum because, at each iteration, placing a job in a particular tier can change the performance of that tier. This affects the *Utility* calculated and the selected tier for each job in all the previous iterations, but the greedy algorithm cannot update those selections to balance the trade-off between cost and performance. For completeness, we compare our approach with two versions of the greedy algorithm: `Greedy exact-fit` attempts to limit the cost by not over-provisioning extra storage space for workloads, while `Greedy over-provisioned`



(a) Normalized tenant utility.



(b) Total monetary cost and runtime.



(c) Capacity breakdown.

Figure 7: Effectiveness of CAST and CAST++ on workloads with reuse, observed for key storage configurations. Note: `Greedy over-prov` represents `greedy over-provisioned`. Tenant utility is normalized to that of the configuration from basic CAST.

will assign extra storage space as needed to reduce the completion time and improve performance.

The tenant utility of `Greedy exact-fit` is as poor as `objStore 100%`. This is because `Greedy exact-fit` only allocates *just enough* storage space without considering performance scaling. `Greedy over-provisioned` is able to outperform `ephSSD 100%`, `persHDD 100%` and `objStore 100%`, but performs slightly worse than `persSSD 100%`. This is because the approach significantly over-provisions `persSSD` and `persHDD` space to improve the runtime of the jobs. The tenant utility improvement under basic CAST is 178% and 113.4%, compared to `Greedy exact-fit` and `Greedy over-provisioned`, respectively.

5.1.3 *Effectiveness for Data Reuse*

CAST++ outperforms all other configurations and further enhances the tenant utility of basic CAST by 14.4% (Figure 7(a)). This is due to the following reasons. (1) CAST++ successfully improves the tenant utility by exploiting the characteristics of jobs and underlying tiers and tuning the capacity distribution. (2) CAST++ effectively detects data reuse across jobs to further improve the tenant utility by placing shared data in the fastest `ephSSD`, since we observe that in Figure 7(c) the capacity proportion under CAST++ of `objStore` reduces by 42% and that of `ephSSD` increases by

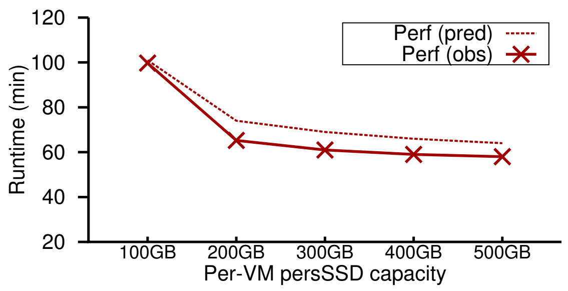


Figure 8: Predicted runtime achieved using CAST's performance scaling regression model vs. runtime observed in experiments by varying the per-VM `persSSD` capacity. The workload runs on the same 400-core cluster as in §5.1.2.

29%, compared to CAST. This is because CAST++ places jobs that share the data on `ephSSD` to amortize the data transfer cost from `objStore`.

5.1.4 *Accuracy of the Regression Model*

Figure 8 compares predicted runtime to observed runtime for a small workload consisting of 16 modest-sized jobs. The total dataset size of all jobs is 2 TB. Both the predicted and the observed runtime follow the same general trend, with an average prediction error of 7.9%, which demonstrates the accuracy of our cubic Hermite spline regression models. The margin of error is tolerable for our case, since the focus of CAST is to help tenants compare and choose among different tiering plans.

5.2 Meeting Workflow Deadlines

5.2.1 *Methodology*

In our next set of experiments, we evaluate the ability of CAST++ to meet workflow deadlines while minimizing cost. We compare CAST++ against four storage configurations without tiering and a fifth configuration from the basic, workflow-oblivious CAST. This experiment employs five workflows with a total of 31 analytics jobs, with the longest workflow consisting of 9 jobs. We focus on large jobs that fully utilize the test cluster's compute capacity.

We consider the completion time of a workflow to be the time between the start of its first job and the completion of its last job. The *deadline* of a workflow is a limit on this completion time, i.e., it must be less than or equal to the deadline. We set the deadline of the workflows between 15 – 40 minutes based on the job input sizes and the job types comprising each workflow. When a job of a workflow completes, its output is transferred to the input tier of the next job. The time taken for this cross-tier transfer is accounted as part of the workflow runtime by CAST++. However, since CAST is not aware of the intra-workflow job dependencies (treating all currently running workflows as a combined set of jobs), CAST cannot account for this transfer cost.

5.2.2 *Deadline Miss Rate vs. Cost*

Figure 9 shows the *miss rate* of workflow deadlines for the studied configurations. The miss rate of a configuration is the fraction of deadlines missed while executing the workflows using that configuration. CAST++ meets all the deadlines and incurs the lowest cost, comparable to that of

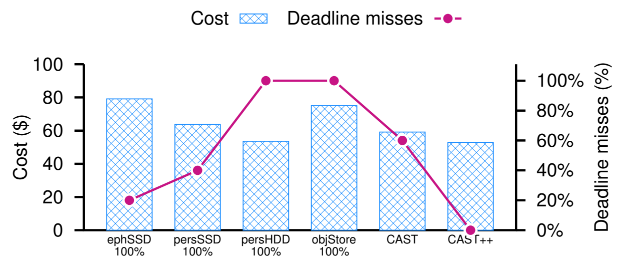


Figure 9: Deadline miss rate and cost of Cast++ compared to Cast and four non-tiered configurations.

`persHDD` that is the lowest-priced but the slowest tier and has a miss rate of 100%.

Cast misses 60% of the deadlines because of two reasons: (1) it selects slow tiers for several jobs in each workflow when trying to optimize for tenant utility; and (2) by not accounting for the cross-tier transfer time, it mis-predicts the workflow runtime. However, Cast incurs a lower cost compared to the non-tiered configurations, because it selects lower-priced tiers for many of the jobs.

Despite being the fastest tier, `ephSSD` misses 20% of the deadlines because of the need to fetch the input data for every workflow from `objStore`. `persSSD` misses 40% of the deadlines because it performs slightly worse than `ephSSD` for I/O intensive jobs. Finally, `objStore` misses all of the deadlines because it is slower than or as fast as `persSSD`. It incurs a higher cost because of the `persSSD`, which is needed for storing intermediate data.

In summary, Cast++ outperforms Cast as well as non-tiered storage configurations in meeting workflow deadlines, and does so while minimizing the cost of running the workflows on the cloud cluster.

6. DISCUSSION

In the following, we discuss the applicability and limitations of our storage tiering solutions.

Analytics Workloads with Relatively Fixed and Stable Computations Analytics workloads are known to be fairly *stable* in terms of the number of types of applications. Recent analysis by Chen et. al. [18] shows that a typical analytics workload consists of only a small number of common computation patterns in terms of analytics job types. For example, a variety of Hadoop workloads in Cloudera have four to eight unique types of jobs. Moreover, more than 90% of all jobs in one Cloudera cluster are `Select`, `PigLatin` and `Insert` [18]. These observations imply that a relatively fixed and stable set of analytics applications (or analytics kernels) can yield enough functionality for a range of analysis goals. Thus, optimizing the system for such applications, as in Cast, can significantly impact the data analytics field.

Dynamic vs. Static Storage Tiering Big data frameworks such as Spark [47] and Impala [11] have been used for real-time interactive analytics, where dynamic storage tiering is likely to be more beneficial. In contrast, our work focuses on traditional batch processing analytics with workloads exhibiting the characteristics identified above. Dynamic tiering requires more sophisticated fine-grained task-level scheduling mechanisms to effectively avoid the straggler issue. While dynamic tiering in our problem domain can help to some extent, our current tiering model adopts a simple yet effective coarse-grained tiering approach. We believe we have provided a *first-of-its-kind* storage tiering methodology for cloud-based analytics. In the future, we plan to enhance Cast to incorporate fine-grained dynamic tiering as well.

7. CONCLUSION

In this paper, we design Cast, a storage tiering framework that performs cloud storage allocation and data placement for analytics workloads to achieve high performance in a cost-effective manner. Cast leverages the performance and pricing models of cloud storage services and the heterogeneity of I/O patterns found in common analytics applications. An enhancement, Cast++, extends these capabilities to meet deadlines for analytics workflows while minimizing the cost. Our evaluation shows that compared to extant storage-characteristic-oblivious cloud deployment strategies, Cast++ can improve the performance by as much as 37.1% while reducing deployment costs by as much as 51.4%.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback that helped us improve the paper. We also thank Taha Hasan for helpful discussions on the idea in its early stages, and Krishnaraj K. Ravindranathan for help in improving the presentation. This work was sponsored in part by the NSF under Grants CNS-1405697 and CNS-1422788.

9. REFERENCES

[1] Amazon EMR. http://aws.amazon.com/elasticmapreduce.
[2] Apache Hadoop. http://hadoop.apache.org.
[3] Apache Oozie. http://oozie.apache.org.
[4] Azure Storage. http://azure.microsoft.com/en-us/services/storage.
[5] EC2 Storage. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Storage.html.
[6] Google Cloud Pricing. https://cloud.google.com/compute/#pricing.
[7] Google Cloud Storage Connector for Hadoop. https://cloud.google.com/hadoop/google-cloud-storage-connector.
[8] Hadoop on Google Compute Engine. https://cloud.google.com/solutions/hadoop.
[9] HDFS-2832. https://issues.apache.org/jira/browse/HDFS-2832.
[10] HP Cloud Storage. http://www.hpcloud.com/products-services/storage-cdn.
[11] Impala. http://impala.io.
[12] Microsoft Azure HDInsight. http://azure.microsoft.com/en-us/services/hdinsight.
[13] Mumak: MapReduce Simulator. https://issues.apache.org/jira/browse/MAPREDUCE-728.
[14] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and C. E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of USENIX ATC 2013*.
[15] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica.

PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of USENIX NSDI 2012*.

[16] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of ACM SoCC 2013*.

[17] S. Babu. Towards automatic optimization of MapReduce programs. In *Proceedings of ACM SoCC 2010*.

[18] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in Big Data systems: A cross-industry study of MapReduce workloads. *PVLDB*, 5(12):1802–1813, Aug. 2012.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of ACM SoCC 2010*.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of USENIX OSDI 2004*.

[21] I. Elghandour and A. Aboulnaga. ReStore: Reusing results of MapReduce jobs. *PVLDB*, 5(6):586–597, Feb. 2012.

[22] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of USENIX FAST 2011*.

[23] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook Messages case study. In *Proceedings of USENIX FAST 2014*.

[24] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[25] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for Big Data analytics. In *CIDR*, 2011.

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

[27] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of ACM SoCC 2012*.

[28] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating Phase Change Memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of USENIX FAST 2014*.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[30] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, 10th edition, August 2011.

[31] Krish K.R., A. Anwar, and A. R. Butt. hatS: A heterogeneity-aware tiered storage for Hadoop. In *Proceedings of IEEE/ACM CCGrid 2014*.

[32] Krish K.R., A. Anwar, and A. R. Butt. øSched: A heterogeneity-aware Hadoop workflow scheduler. In *Proceedings of IEEE MASCOTS 2014*.

[33] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace. WOHA: Deadline-aware Map-Reduce workflow scheduling framework over Hadoop clusters. In *Proceedings of IEEE ICDCS 2014*.

[34] Z. Li, A. Mukker, and E. Zadok. On the importance of evaluating storage systems' $costs. In *Proceedings of USENIX HotStorage 2014*.

[35] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for MapReduce workflows. *PVLDB*, 5(11):1196–1207, July 2012.

[36] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of ACM/IEEE SC 2011*.

[37] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of USENIX ATC 2014*.

[38] M. Mihailescu, G. Soundararajan, and C. Amza. MixApart: Decoupled analytics for shared storage systems. In *Proceedings of USENIX FAST 2013*.

[39] K. P. Puttaswamy, T. Nandagopal, and M. Kodialam. Frugal storage for cloud file systems. In *Proceedings of ACM EuroSys 2012*. ACM.

[40] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of IEEE MSST 2010*.

[41] A. Verma, L. Cherkasova, and R. H. Campbell. Play it again, SimMR! In *Proceedings of IEEE CLUSTER 2011*.

[42] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in MapReduce setups. In *Proceedings of IEEE MASCOTS 2009*.

[43] H. Wang and P. Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of USENIX FAST 2014*.

[44] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *Proceedings of USENIX NSDI 2012*.

[45] D. Yuan, Y. Yang, X. Liu, and J. Chen. A data placement strategy in scientific cloud workflows. *Future Generation Computer Systems*, 26(8):1200 – 1214, 2010.

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

[47] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX NSDI 2012*.