Cake: Enabling High-level SLOs on Shared Storage Systems

Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, Ion Stoica
University of California, Berkeley
{awang, shivaram, alspaugh, randy, istoica}@cs.berkeley.edu

Abstract

Cake is a coordinated, multi-resource scheduler for shared distributed storage environments with the goal of achieving both high throughput and bounded latency. Cake uses a two-level scheduling scheme to enforce high-level service-level objectives (SLOs). First-level schedulers control consumption of resources such as disk and CPU. These schedulers (1) provide mechanisms for differentiated scheduling, (2) split large requests into smaller chunks, and (3) limit the number of outstanding device requests, which together allow for effective control over multi-resource consumption within the storage system. Cake's second-level scheduler coordinates the first-level schedulers to map high-level SLO requirements into actual scheduling parameters. These parameters are dynamically adjusted over time to enforce high-level performance specifications for changing workloads. We evaluate Cake using multiple workloads derived from real-world traces. Our results show that Cake allows application programmers to explore the latency vs. throughput trade-off by setting different high-level performance requirements on their workloads. Furthermore, we show that using Cake has concrete economic and business advantages, reducing provisioning costs by up to 50% for a consolidated workload and reducing the completion time of an analytics cycle by up to 40%.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes

General Terms

Performance, Measurement, Design

Keywords

Consolidation, Storage Systems, Service-Level Objectives, Two-level Scheduling

1. INTRODUCTION

Datacenter applications can be grouped into two broad classes: user-facing, latency-sensitive front-end applications, and internal, throughput-oriented batch analytics frameworks. These applications access distributed storage systems like HBase [2], Cassandra [3], and HDFS [1]. Storage systems are typically not shared between these classes of applications because of an inability to multiplex latency-sensitive and throughput-oriented workloads without violating application performance requirements. These performance requirements are normally expressed as *service-level objectives* (SLOs) on throughput or latency. For instance, a web client might require a 99^{th} percentile latency SLO of 100ms for key-value writes, and a batch job might require a throughput SLO of 100 scan requests per second. SLOs reflect the performance expectations of end users, and Amazon, Google, and Microsoft have identified SLO violations as a major cause of user dissatisfaction [32, 24].

To satisfy the SLOs of both latency-sensitive and throughput-oriented applications, businesses typically operate separate, physically distinct storage systems for each type of application. This has significant economic costs. First, separate storage systems must each be provisioned individually for peak load. This requires a higher degree of *overprovisioning* and contributes to *underutilization* of the cluster,

SOCC'12, October 14-17, 2012, San Jose, CA USA

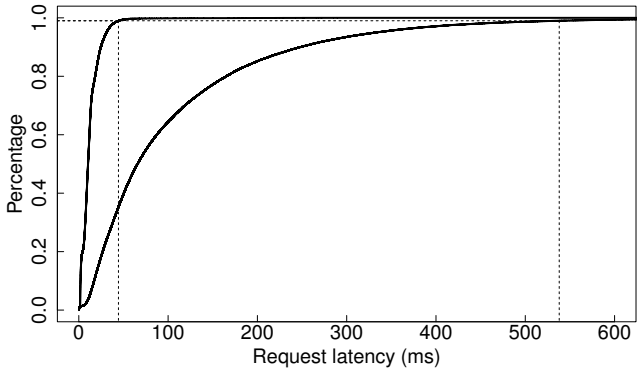


Figure 1: A latency-sensitive front-end application's 99th percentile latency increases from 44ms to 538ms (~12x) when run concurrently with a throughput-oriented batch application. This is because the naïve consolidation strategy results in unmediated resource contention within the storage system.

both problems in datacenter environments where cost is a major consideration [6]. Second, segregation of data leads to *degraded user experience* from a delayed processing cycle between front-end and back-end systems. For instance, computing the "friends-of-friends" relation on Facebook might require loading the current friend list into a back-end storage system, performing the friends-of-friends analysis, and then loading the results back into the front-end storage system. The necessity of copying data back and forth prolongs the analytics cycle, directly leading to degraded user experience [7]. Finally, there is increased *operational complexity* from requiring additional staff and expertise, and greater exposure to software bugs and configuration errors [25].

Consolidating latency-sensitive and throughput-oriented workloads onto a single storage system solves these problems. Ideally, both classes of workloads could be consolidated without sacrificing the dual goals of low latency and high throughput. Latency-sensitive and throughput-oriented requests would both receive same performance as if they were running in isolation on a dedicated system.

However, if a system is not aware of each client's performance requirements, a naïve approach to consolidation results in unacceptable SLO violations, especially for high-percentile latency SLOs. An example of this is quantified in Figure 1, where we see that 99th percentile HBase latency increases from 44ms to 538ms (~12x) with a naïve consolidation scheme. Furthermore, there exists a fundamental trade-off between throughput and latency with rotational media that makes it impossible to simultaneously achieve both low latency and high throughput. Achieving low latency requires that request queues within the storage system remain short, while achieving high throughput requires long queues to maximize utilization and minimize disk head movement.

Many existing solutions for the storage layer operate within the hypervisor or storage controller, and focus solely on controlling disk-level resources [23, 29, 21]. However, there is a disconnect between the high-level SLOs specified for distributed storage system operations and disk-level scheduling and performance parameters like MB/s and IOPS. Translating between the two requires tedious, manual tuning by the programmer or system operator. Furthermore, providing high-level storage SLOs require consideration of resources beyond the disk. Distributed storage systems perform CPU and memory intensive tasks like decompression, checksum verification, and deserialization. These tasks can significantly contribute to overall request latency, and need to be considered as part of a multi-resource scheduling paradigm.

In this paper, we present Cake, a coordinated, two-level scheduling approach for shared storage systems. Cake takes a more holistic view of resource consumption, and enables consolidation of latency-sensitive and batch workloads onto the same storage system. Cake lets users specify their performance requirements as high-level, end-to-end SLOs on storage system operations, and does not require manual tuning or knowledge of low-level system parameters. This is done with a two-level scheduling approach. *First-level* schedulers are placed at different points within the system to provide effective control over resource consumption. As part of this, we identify and implement three precepts that are essential for effective scheduling at a resource: (1) provide mechanisms for differentiated scheduling, (2) split large requests into smaller chunks, and (3) limit the number of outstanding device requests. Cake's *second-level* scheduler is a feedback loop which continually adjusts resource allocation at each of the first-level schedulers to maximize SLO compliance of the system while also attempting to increase utilization.

We applied the ideas from Cake to HBase, a distributed storage system that operates on top of HDFS, a distributed file system. We evaluated Cake using real-world front-end and batch workload traces, and found that Cake's two-level scheduling approach is effective at enforcing 99th percentile latency SLOs of a front-end running simultaneously alongside a throughput-oriented batch workload. This allows users to flexibly move within the latency vs. throughput trade-off space by specifying different high-level SLOs on their workloads. We also demonstrate that Cake has concrete economic and business benefits. By consolidating batch and front-end workloads onto a shared storage system, Cake can reduce provisioning costs by up to 50% and can reduce the completion time of an analytics processing cycle by up to 40%.

2. RELATED WORK

Much prior work has examined issues related to sharing of storage systems, service-level objectives, and multi-resource scheduling. We address how these bodies of work differ from Cake.

Block-level approaches. Many solutions in this area operate at the block-level by modifying the storage controller, the hypervisor, or the operating system. These solutions differ from Cake in that they only provide block-level guarantees, only consider disk resources, and tend to focus on throughput rather than high-percentile latency. Argon provides performance isolation for shared storage clients using cache partitioning, request amortization, and quanta-based disk time scheduling [37]. PARDA enables fair sharing among distributed hosts using a FAST TCP-like mechanism to detect congestion on a shared storage array [21]. However, these systems focus on providing guarantees on disk throughput, and PARDA concerns itself only with fair sharing, not absolute performance SLOs. mClock and Maestro use adaptive mechanisms to enforce storage SLOs in the context of virtual machines and storage arrays respectively [23, 29]. These systems provide latency guarantees, but not for 99th percentile latency. There are also many other systems which focus on fair sharing and quality-of-service in the storage context [10, 33, 27, 20, 9, 30].

Multi-resource scheduling. Related work has also examined the domain of multi-resource scheduling for fairness and performance. Databases bear many similarities in that they also seek to provide high-level guarantees on storage operations, but differ from our work in the applied techniques, goals, and setting. Soundararajan et al. used sampling to build application cache and disk performance models, and used these models to determine efficient disk bandwidth and cache partitioning [35]. Cake differs from this work in that it ties resource allocation back to client SLOs, focuses on high-percentile latency, and attempts to do this for a layered software architecture, not a monolithic database.

Soundararajan and Amza and Padala et al. share our goal of providing application-level SLOs for layered software architectures [34, 31]. However, these approaches require making modifications to the kernel or hypervisor, whereas our approach operates purely from userspace.

Dominant Resource Fairness (DRF) provides a generalization of fair scheduling to multiple resources [19, 18]. Although not directly relevant to Cake, DRF could potentially be used as one of the underlying mechanisms used to adjust end-to-end performance of clients interfacing with Cake.

Distributed storage systems. A number of papers have examined techniques for automatically scaling, provisioning, and load balancing distributed storage system [5, 36, 16, 17, 22, 28, 39]. Some distributed systems, such as Dynamo, have explored the use of quorum-based request replication to reduce 99th percentile latency [4, 13]. Many other distributed storage systems have also been designed for predictable low latency [12, 11]. These ideas and techniques are complementary to our work with Cake, which focuses on providing predictable performance on individual storage nodes in a distributed storage system.

SLOs in the datacenter. Connecting resource allocation to SLOs takes place at many places within the datacenter. Jockey uses feedback control and offline performance models to enforce latency SLOs on MapReduce-style jobs [14]. D^3 uses deadline information to allocate bandwidth among network flows more efficiently [40]. DeTail tries to reduce high-percentile latency for network flows [41].

3. BACKGROUND

With the growing popularity of rich web applications, high-percentile latency SLOs on storage operations are becoming increasingly important. This is because rich web applications display a *request fan-out* pattern, where rendering a single page might require making making many requests to the storage system. In this scenario, a single slow storage request can dominate the overall response time of the page load. For example, constructing a page of search results might involve making 10 requests in parallel to the storage system. If the probability of a slow storage request is 10%, the probability of a slow overall response is 65%. If the probability of a slow storage request can be reduced to 1%, the probability of a slow overall response falls to 10%. Dealing with the latency tail present at the 95th or 99th percentile is thus one of the more important considerations of a user-facing storage system [4, 13].

High-level SLOs are specified by three parameters: the *priority* of the client sending the request, the client's *performance goal*, and a *throughput contract* that specifies the amount of load that will be sent by the client. Performance goals are specified in terms of either *percentile latency* or *average throughput*. A client could ask for 99th percentile latency of 100ms on get requests, or ask for a throughput of 50 scan requests a second. The throughput contract prevents a client from starving other clients by flooding the system with requests. If a client sends more load than its throughput contract, the extra load can be handled without any performance guarantees in a best-effort fashion.

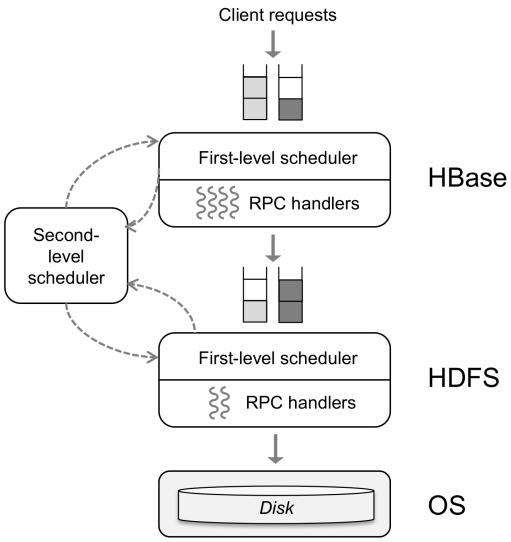


Figure 2: Architecture of the Cake software stack on a single storage node. Cake adds first-level schedulers to the RPC layers of HBase and HDFS. The first-level schedulers are coordinated by Cake's SLO-aware second-level scheduler.

Cake addresses the problem of enforcing SLOs in a shared storage environment. To do this, we make the following simplifying assumptions:

1. We consider a simplified workload model with just two classes of requests: small, high-priority, latency-sensitive requests issued by a front-end application and large, low-priority, throughput-oriented requests issued by a batch application. This captures the two broad categories of real-life datacenter workloads, and examines the problem of consolidating latency-sensitive and throughput-oriented workloads.

2. We do not deal with the problem of *SLO admission control*, which requires the system to judge if a workload's SLO is attainable based on provisioned hardware resources and the SLOs of already admitted workloads. Instead, we assume that the system is provisioned to meet the throughput contract required by the front-end application, such that in the limit, the front-end application's SLO can be met by running no load from the batch application.

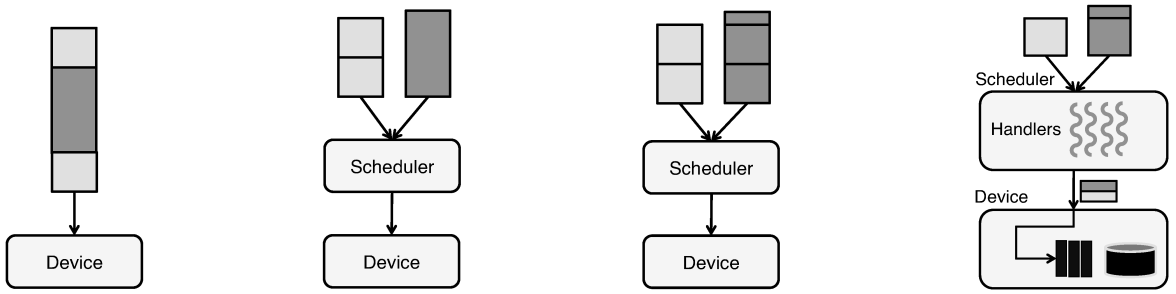
High-level SLO specification in Cake is thus simplified as follows. The front-end client is set to high priority, and specifies a percentile latency performance goal. We assume that it is possible for this performance goal and throughput contract to be met by the system. The batch client is set to low priority, and specifies a very high average throughput target. This means the batch client is allowed to use as much capacity of the system as is possible without impacting the front-end client's performance.

Making these assumptions allows us to demonstrate Cake's ability to enforce SLOs of realistic consolidated workloads without being hindered by the full scope of providing performance guarantees for storage systems.

4. ARCHITECTURE

Storage systems are often designed as a number of software components that communicate over RPC interfaces. This is true for the architecture of HBase, a distributed storage system that builds on top of HDFS, a distributed filesystem. HBase lets client applications make single and multi-row requests on columns in a table. When a client request arrives at an HBase slave, the request waits in a queue until it is assigned an RPC handler. The HBase handler then parses the request and does an index lookup to locate the HDFS slave with the requested data. HBase then issues a series of RPCs to the HDFS slave, which is usually colocated on the same physical machine. The requests to HDFS again wait in a queue until handled, at which point HDFS uses standard file I/O methods to read the data off disk. This data is returned to the HBase slave, and then finally to the client application.

This presents a clean separation of resource consumption during request processing. The distributed storage system (HBase) is responsible for CPU and memory intensive operations like maintaining an index, caching rows, performing checksum verifications, and decompressing stored data. The distributed filesystem (HDFS) is essentially a thin layer over the block device, and is responsible for I/O operations involving the disk. Because these software components interact solely over RPC, the RPC layer at each component is a powerful and practical place to do resource scheduling.



(a) Using a single, shared FIFO queue leads to head-of-line blocking of high-priority requests.

(b) Lack of preemption means large requests can block access to a resource when scheduled.

(c) Chunking large requests enables some preemption, but can lead to decreased throughput.

(d) Limiting the number of outstanding requests reduces resource contention, but can also limit performance.

Figure 3: Many issues arise when attempting to do request scheduling at a resource. First-level schedulers at each resource need to support differentiated scheduling, chunking of large requests, and limiting the number of outstanding requests to be effective.

Cake's two-level scheduling scheme exploits the clean separation of resource consumption in systems like HBase (Figure 2). We start by identifying three properties that are necessary for effective scheduling at a single resource. Cake provides these properties by adding a *first-level* scheduler to the RPC layer of each software component in the storage system. Next, we generalize to the multi-resource situation. Cake's *second-level* scheduler coordinates allocation at each of the first-level schedulers to enforce high-level SLOs of a consolidated front-end and batch workload.

4.1 First-level Resource Scheduling

Resource consumption at a single resource can be modeled as a software component that issues requests to a hardware device. The software component issues multiple concurrent requests to the device as part of processing a higher-level operation like a key-value read or scan. These requests potentially wait in an on-device queue to be processed, and the device can potentially process multiple requests in parallel.

We identified three scheduler properties that are necessary for effective first-level resource scheduling.

Differentiated scheduling. Figure 3(a) depicts the problem with putting all incoming requests into a single FIFO queue. High-priority requests can suffer from head-of-line blocking when they arrive behind low-priority requests. This is solved by separating requests to each resource into different queues based on priority (Figure 3(b)). The first-level scheduler chooses amongst these queues based on proportional shares and reservations set by the second-level scheduler [38]. Here, a reservation means assigning a client a guaranteed slice of a resource by setting aside either an execution unit or an outstanding I/O slot.

Split large requests. Even with different queues, large requests can still block small requests if the hardware resource is not *preemptible*. Here, when a large request is scheduled, it holds the resource until it completes. Requests can be non-preemptible because of restrictions of the hardware resource (e.g. I/O issued to the disk controller), or the structure of the software component (e.g. a blocking thread pool).

Because both of these circumstances arise in storage systems, the first-level scheduler needs to split large requests into multiple smaller chunks (Figure 3(c)). This allows for a degree of preemption, since latency-sensitive requests now only need to wait for a small chunk rather than an entire large request. However, deciding on an appropriate chunk size presents a trade-off between latency and throughput. Smaller chunks allow finer-grained preemption and better latency, but result in lower throughput from increased disk seeks and context switches.

Control number of outstanding requests. A resource can typically process some number of concurrent requests in parallel (Figure 3(d)). This number should be chosen to limit queueing at lower levels, where scheduling decisions are arbitrarily made outside of Cake by the OS or disk controller. With too many outstanding requests, queuing will increase at lower levels. With too few outstanding requests, the hardware device is underutilized.

It is not sufficient to choose the number of outstanding requests based on hardware properties such as the number of disk spindles or the number of CPU cores. This is because requests can differ greatly in their resource consumption. A resource might be able to admit many small requests without becoming oversubscribed, but a few large requests might saturate the same resource. The level of concurrency needs to be adjusted dynamically to straddle under- and overload based on the composition of the workload.

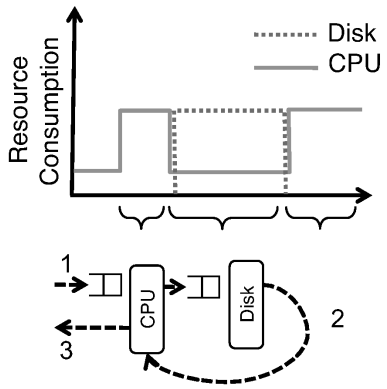


Figure 4: Processing a single storage system request can lead to concurrent usage of multiple resources. Enforcing high-level latency SLOs requires coordinated scheduling at each resource.

SLO performance	Action	Justification
perf $\gg$ SLO	$\downarrow$ disk alloc	SLO safely met
perf < SLO	$\uparrow$ disk alloc	SLO is not met
Queue occ. at r	**Action**	**Justification**
$r >$ disk	$\uparrow$ alloc at r	bottleneck at r, not disk
$r \ll$ disk	$\downarrow$ alloc at r	r is underutilized

Table 1: Scheduling rules for the SLO compliance-based and queue occupancy-based second-level scheduling phases. SLO compliance-based changes apply to disk. Queue occupancy-based changes apply to non-disk resources.

To address this, we borrow ideas from TCP congestion control. TCP Vegas [8] uses packet RTT measurements as an early indicator of congestion in the network. Higher than normal RTTs are indicative of long queues at intermediate switches within the network. Applied to our situation, network congestion is analogous to oversubscription of a resource. In both cases, latency increases when additional queuing happens in lower levels.

The first-level scheduler dynamically adjusts the number of outstanding requests based on measured device latency. As in TCP, we use an additive-increase, multiplicative-decrease (AIMD) policy to adjust this number. AIMD provably converges in the general case [26]. This method is described in further detail in §5.2.

4.2 Second-level Scheduling

Next, we examine the challenges of scheduling in the multi-resource case, and provide an overview of how Cake's second-level scheduler coordinates resource allocation at each first-level scheduler to enforce high-level SLOs. Further implementation-specific details of second-level scheduling are provided in §5.3.

4.2.1 Multi-resource Request Lifecycle

Request processing in a storage system involves far more than just accessing disk, necessitating a coordinated, multi-resource approach to scheduling. Figure 4 depicts an example of multi-resource request processing in a storage system. A read request arrives at a storage node where it waits to be admitted (1). Once admitted, the read request uses CPU to deserialize the request body and find the location of the requested data on disk. Next, the read request issues a series of reads to the disk, using the time between disk requests to decompress data in a streaming fashion with the CPU (2). Finally, it uses a burst of CPU to verify the integrity of the data, and then to compress, serialize and send a response (3).

When request processing requires using multiple resources, end-to-end latency is affected by dependencies between stages of request processing and also resource contention within the system. A single stage suffering from resource contention can starve subsequent stages of processing of useful work. If this bottleneck can be identified, reducing resource contention at this stage will improve performance, but also shifts the bottleneck elsewhere. Improving overall performance thus requires coordinated measurement and scheduling at multiple resources.

4.2.2 High-level SLO Enforcement

Scheduling policies can be designed to achieve different goals: high throughput, real-time deadlines, proportional fairness, max-min fairness, etc. Cake's second-level scheduler is designed to first satisfy the latency requirements of latency-sensitive front-end clients, then maximize the throughput of throughput-oriented batch clients. The second-level scheduler does this by collecting performance and utilization metrics at each resource, and then using this information to make resource allocation decisions at each of the first-level schedulers. This forms a feedback loop that lets the second-level scheduler adapt to changes in the workload. Second-level scheduling decisions are made in two phases: first for disk in the SLO compliance-based phase and then for non-disk resources in the queue occupancy-based phase.

The *SLO compliance-based phase* adjusts disk scheduling allocation based on the performance of the front-end client. This is because I/O queue time tends to dominate overall latency in storage systems, so meaningful changes in end-to-end latency must be driven by changes

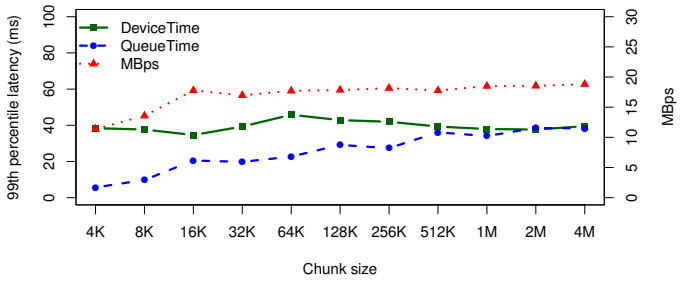


Figure 5: We measured the queue time, device processing time, and throughput of a constant workload with different chunk sizes. Device time is dominated by seek latency, and remains fairly constant. Queue time and throughput increase as the chunk size increases. To achieve low latency at high percentiles, we split requests into smaller chunks.

in disk scheduling allocation [21]. If the latency-sensitive client fails fails to meet its SLO target over the previous time interval, the second-level scheduler increases the latency-sensitive client's disk allocation. Conversely, if the latency-sensitive client is safely meeting its SLO, the second-level scheduler reduces the latency-sensitive client's disk allocation to increase batch throughput.

The *queue occupancy-based phase* balances allocation at other resources to make full use of a client's disk allocation. This is necessary because of the inter-stage dependencies described in §4.2.1; a client could be incorrectly throttled at a non-disk resource. *Queue occupancy* is defined as the percentage of time over the scheduling interval that at least one request from the client was present in the queue. For instance, a queue occupancy of 100% means that there was always a request from the client present in the queue over the interval. Note that low absolute queue occupancy is not necessarily indicative of over-allocation. If a client has a bursty arrival pattern, its queue occupancy might never rise above a small percentage. However, it might still require full use of its allocation during a burst to meet its SLO.

Examining queue occupancy at each resource allows us to identify bottlenecks and incorporate the nature of the workload into our scheduling decisions. We size allocations at non-disk resources using *relative* comparisons of queue occupancy. If a resource's queue occupancy is the highest across all resources, its allocation is increased additively. This indicates that the client is bottlenecked on this resource rather than disk, and that the disk allocation is not being fully utilized. Alternatively, if a resource's queue occupancy is substantially lower than the disk's queue occupancy, its allocation is decreased additively. This indicates that the resource is underutilized and allocation can be decreased. The former condition indicates that the resource is underutilized, while the latter indicates that processing is still disk-bound (which is the desired outcome).

Together, the two phases of second-level scheduling harmonize to enforce high-level SLOs (summarized in Table 1). The initial SLO compliance-based phase decides on disk allocations based on client performance. The queue occupancy-based phase balances allocation in the rest of the system to keep the disk utilized and improve overall performance.

5. IMPLEMENTATION

We applied the Cake multi-resource scheduling model to a concrete implementation on HBase 0.90.3 and HDFS 0.20.2 (Figure 2). We interposed at the RPC layers in both HDFS and HBase, adding new per-client queues and a first-level scheduler which provides the three properties described in §4.1. Client requests were separated out into different queues based on a "client name" field in the HBase and HDFS RPC format. HBase and HDFS were also modified to use a dynamically-sizable thread pool rather than a fixed-size thread pool or spawning a new thread per incoming request.

We start by describing how we implemented the three first-level scheduling properties from §4.1 at HBase and HDFS. Implementing the first-level schedulers required choosing appropriate values for some system parameters; namely, the chunking factor for large requests at HDFS and the lower and upper latency bounds used to dynamically size the number of outstanding requests at HBase. We briefly discuss the sensitivity of these parameters to our tested system configuration of c1.xlarge instances on Amazon EC2 with 8 CPU cores, 7GB of RAM, and 4 local disks.

Finally, we describe the operation of the two second-level scheduling phases outlined in §4.2.2. We cover how the second-level scheduler adjusts shares and reservations at the first-level schedulers based on measured performance and queue occupancy, and the steps taken to improve stability of the algorithm, avoid under-utilization, and avoid bottlenecks on non-disk resources.

5.1 Chunking Large Requests

The Cake first-level scheduler at HDFS splits large read requests into multiple, smaller chunks. Splitting is not performed at HBase because it would have required extensive modification of its thread management. In our evaluation in §6, we found that splitting at HDFS alone was sufficient because disk access tends to dominate end-to-end latency.

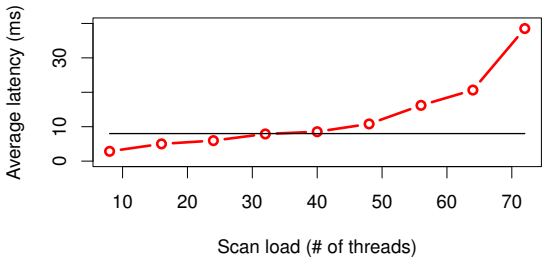


Figure 6: Front-end client latency in HBase increases sharply after concurrent batch load increases above 40. To avoid this region, we chose conservative lower and upper latency bounds of 6ms and 8ms for sizing the HBase thread pool. The 8ms upper bound is plotted.

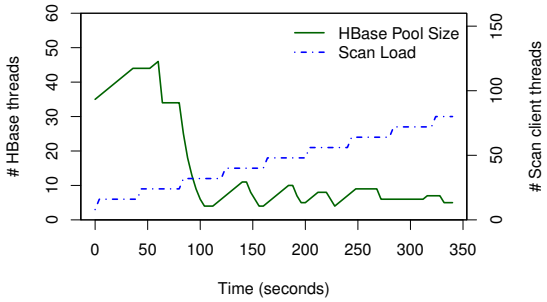


Figure 7: HBase thread pool size initially increases as the system warms up, but decreases as the batch workload steps up and CPU contention increases. The pool size begins to converge at $t = 100$, ultimately settling at $\tilde{8}$ threads by the end of the experiment.

To determine the proper chunk size in HDFS, we measured how disk device time, queue time, and throughput varies with the chunk size using the `blktrace` tool (Figure 5). We ran a constant batch workload issuing 4MB read requests to HDFS, which were then split into smaller chunks at the specified chunk size. We plot the 99th percentile latency of device processing and queue times for the chunked requests, and the average throughput.

Note that queue time increases as the chunk size increases. The device processing time remains fairly constant since 99th percentile latency is dominated by seek time, not rotational latency. Per-request throughput plateaus after 16KB.

Based on this, we chose to use a chunk size of 64KB for HDFS. This is because the throughput of an entire 4MB read request is affected when it is split into more chunks at smaller chunk sizes. Even though a 4KB chunk size has excellent queue time, it requires 16 times more I/Os than a 64KB chunk size. This makes the sum of queue time and device time much higher at small request sizes.

We believe that 64KB is an appropriate chunk size for most rotational storage media, since seek time will always be an important factor for 99th percentile latency. Solid-state storage could likely support a larger chunk size because of its more efficient random read performance, but we have not tested this configuration.

5.2 Number of Outstanding Requests

We modified HBase and HDFS to use a dynamically-sized thread pool for processing requests. Each thread is responsible for a single request at a time. The size of each thread pool is adjusted dynamically to minimize resource contention at each resource, which keeps processing time constant.

HBase uses the dynamic sizing mechanism based on TCP Vegas described in §4.1. Every ten seconds, Cake examines the average processing latency within HBase over the last interval for single-row get requests. If the latency is below 6ms, it additively increases the size of the thread pool by 1. If the latency is above 8ms, it multiplicatively decreases the size of the thread pool by 25%.

We determined these lower and upper latency bounds empirically by measuring how the latency of a constant front-end client workload changes as batch load increases (Figure 6). Batch load was increased by adding additional batch client threads making scan requests, without a throughput limit. We see that front-end latency increases sharply after 40 threads of scan load. 6ms and 8ms were conservative bounds chosen to avoid this region.

In Figure 7, we show how the dynamic thread pool sizing algorithm responds as batch load changes. We allowed HBase to dynamically vary its thread pool size between 4 and 72 based on the above policy. An additional 8 batch client threads were turned on every 40 seconds. The thread pool size initially increases as the system warms up, but begins to decrease as batch load increases. After $t = 100$, the pool size begins to converge due to AIMD adjustments, ultimately settling on a pool size of approximately 8 at maximum batch load.

These latency bounds will potentially have to be readjusted for different system configurations, as the configuration affects the baseline processing latency of the single-row get requests we use to estimate CPU contention. With faster processors and HBase software improvements, the appropriate latency bounds might be lower. However, slightly reducing the bounds is unlikely to meaningfully change 99th percentile latency, especially if the workload remains disk-bound.

For HDFS, we found that a simple pool sizing policy based on soft limits worked well. This is because chunking large requests already gives Cake adequate scheduling control over disk access. We set a soft-limit on the HDFS thread pool size of 6, which is approximately equal to the number of disks in our system. The pool size is allowed to temporarily increase by one when a thread blocks on a non-disk operation, such as reading or writing to a network socket. This improves utilization while still enforcing the soft-limit on the number of threads

Algorithm 1 SLO compliance-based scheduling phase

```
1: function SLOSCHEDULING( )
2:     Client l is latency-sensitive client
3:     Client b is batch client
4:     if l.perf < 1.0 then                                          ▷ SLO not met
5:         if l.hdfs_share > 99% and more handlers then
6:             Give (1 / shared pool size) of share to b
7:             Reserve additional HDFS handler for l
8:         else
9:             Give LinearModel() share from b to l
10:    else if l.perf > 1.2 then                                     ▷ SLO exceeded
11:        if l.hdfs_share < 1% and l has a reservation then
12:            Release one of l's reserved HDFS handlers
13:            Take (1 / shared pool size) of share from b
14:        else
15:            Give LinearModel() share from l to b
16: function LINEARMODEL( )
17:    target ← min(l.hdfs_share/l.perf, 100%)
18:    change ← absVal(l.hdfs_share − target)
19:    bounded ← min(change, l.hdfs_share ∗ 10%)
20:    return bounded
```

Algorithm 2 Queue Occupancy-based scheduling phase

```
1: function QUEUEOCCUPANCYSCHEDULING( )
2:     Client l is latency-sensitive client
3:     Client b is batch client
4:     l.hbase_share ← l.hdfs_share
5:     b.hbase_share ← b.hdfs_share
6:     if l.hdfs_occ < l.hbase_occ then
7:         if more handlers then
8:             Reserve an additional HBase handler for l
9:     else if l.hdfs_occ > 1.5 ∗ l.hbase_occ then
10:        if l.hbase_reservation > l.hdfs_reservation then
11:            Release one of l's reserved HBase handlers
```

concurrently accessing disk. Using a TCP-like scheme for determining the HDFS pool size could allow for better throughput by accurately probing the queue depth on each disk, but our evaluation in §6.6 shows that the soft-limit policy still achieves reasonable throughput.

5.3 Cake Second-level Scheduler

The second-level scheduler collects performance and queue occupancy metrics from HBase and HDFS. Every 10 seconds, it uses these metrics to decide on new scheduling allocations at each resource. This allocation happens in two phases. First, in the SLO compliance-based phase, it adjusts allocations at HDFS based on how the latency-sensitive client's performance over the last interval compares to its stated SLO. Next, in the queue occupancy-based phase, it balances HBase allocation based on the measured queue occupancy at HDFS and HBase. We describe in detail the operation of these two scheduling phases.

5.3.1 SLO Compliance-based Scheduling

Each client's performance is normalized based on its SLO target to derive the client's *normalized SLO performance*. For instance, if a client specifies a latency SLO on gets of 100ms but is currently experiencing latency of 50ms, its normalized performance is 2.0.

The second-level scheduler's SLO compliance-based allocation phase is described in Algorithm 1. The second-level scheduler adjusts HDFS allocation if the latency-sensitive client's normalized performance is significantly below (lines 4-9) or above (lines 10-15) the performance target specified in its SLO. The lower bound was necessarily chosen at 1.0 because it indicates that the client is not meeting its SLO and needs additional allocation. The upper bound was chosen conservatively at 1.2. This lets other clients use excess allocation while also

preventing the scheduler from prematurely releasing allocation due to small fluctuations in performance. We found that this upper bound worked well in practice and was not a meaningful tuning parameter for latency SLO compliance.

A simple linear performance model is used to make adjustments to each client's proportional share (lines 16-20). In both the $perf < 1.0$ and $perf > 1.2$ cases, the linear model targets an allocation to achieve a normalized performance of 1.0. We also bound the actual change in allocation to at most 10% of the client's current share. This is because the actual behavior of the system is non-linear, but can be approximated by a linear model over restricted regions. Bounding the step size has the potential effect of increasing convergence time, but is necessary to improve the stability of the algorithm.

Reservations are a stronger mechanism than proportional share, and are set if shares are found to be insufficient. If a client's $perf < 1.0$ and share exceeds 99%, the scheduler will allocate a reserved handler thread from the shared handler pool and take away a proportional amount of share (lines 6-7). For instance, if a client reserves one of two handlers in the shared pool, it releases 50% of share. If $perf > 1.2$ and share is less than 1%, the scheduler will try to return a reserved handler to the shared pool and again compensate with a proportional amount of share (lines 12-13). For instance, if a client releases a reserved handler to bring the shared pool up to three handlers, its proportional share is increased by 33%. Cake prevents the last shared handler at HDFS from being reserved to prevent indefinitely blocking control messages such as schema updates or heartbeat responses.

The second-level scheduler will thus adjust both proportional shares and reservations at the first-level scheduler at HDFS when making changes in allocation. It tries to first satisfy the SLO through use of the work-conserving proportional share mechanism, but will resort to assigning reservations if share alone is insufficient. The use of both of these mechanisms is important for handling different types of workloads; this will be examined further in the evaluation in §6.1.

5.3.2 Queue Occupancy-based Scheduling

The queue occupancy-based scheduling phase sizes the latency-sensitive client's HBase allocation to fully utilize the client's HDFS allocation, while also attempting to avoid underutilization at HBase. The goal is to quickly balance HBase allocation against the current HDFS allocation so that changes made in the SLO compliance-based phase are reflected in end-to-end SLO performance. The intuition for comparing the queue occupancy at different resources was provided in §4.2.

Queue occupancy-based allocation is described in Algorithm 2. An initial attempt at balancing allocation is made by setting HBase shares equal to HDFS shares (lines 4-5) and ensuring that the latency-sensitive client's HBase reservation is not reduced below its HDFS reservation (line 10). However, if an imbalance is detected, the second-level scheduler will take further action by adjusting the latency-sensitive client's HBase reservation. If the client's queue occupancy at HDFS is lower than at HBase, this indicates that the client is incorrectly bottlenecked at HBase. The second-level scheduler will increase the client's HBase allocation to correct this (lines 6-8). Conversely, if queue occupancy at HDFS is significantly greater than occupancy at HBase, the client's HBase allocation is decreased (lines 9-11). Cake also prevents the last handler at HBase from being reserved to avoid blocking control messages like schema changes or replication updates.

The coefficient of 1.5 on line 10 used for relative queue occupancy comparisons was chosen conservatively. Reducing this coefficient could potentially improve batch throughput. However, our evaluation shows that Cake still achieves reasonable batch throughput when running a consolidated workload, and that the queue occupancy-based phase does step reservation down as the workload varies. We did not find this coefficient to be a significant tuning parameter in our experiments.

6. EVALUATION

In our evaluation, we start by first examining the dual utility of both proportional shares and reservations for different workload scenarios. Then, we demonstrate the need for coordinated, multi-resource scheduling to meet end-to-end 99th percentile latency targets. Next, we evaluate the behavior of the second-level scheduler's ability to adapt to a range of challenging consolidated workload scenarios and different latency SLOs. Finally, we demonstrate how consolidating front-end and batch workloads with Cake can lead to significant reductions in total analytics time, improving utilization while still meeting specified front-end latency SLOs.

As stated in §3, we make a number of simplifying assumptions with Cake. Our evaluation focuses on the setting of a single front-end workload with a 99th percentile SLO contending with a lower priority batch workload. While this is sufficient for many real-world applications, extending Cake to enforce multiple, potentially contending SLOs is a direction of potential future work (§7).

Cake supports enforcement of throughput SLOs, which is useful in shared batch processing environments. However, throughput SLOs are significantly easier to enforce than latency SLOs, and this problem has been examined before in prior work (§2). We omit these results for space reasons.

All experiments were run on an Amazon EC2 cluster using `c1.xlarge` instances. These instances had 8 CPU cores and 7GB of RAM. HDFS was configured to use the four local disks on each instance. Unless otherwise noted, we used Yahoo! Cloud Serving Benchmark (YCSB) clients to generate simulated front-end and batch load. Front-end clients were configured to make single-row requests for 8KB of data, with a variety of workload patterns. Batch clients were configured to make 500-row requests for 4MB of data, with unthrottled throughput.

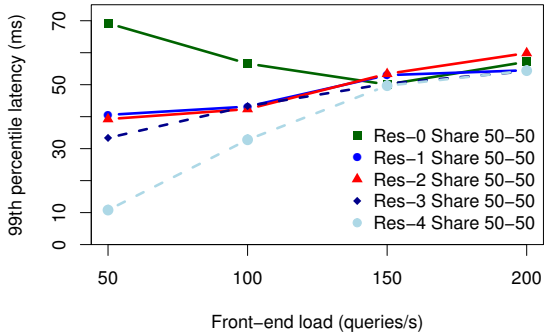


Figure 8: When the front-end client is sending low throughput, reservations are an effective way of reducing queue time at HDFS. Proportional share is insufficient because it will run batch requests when the front-end queue at HDFS is empty, forcing newly arriving front-end requests to wait in the queue.

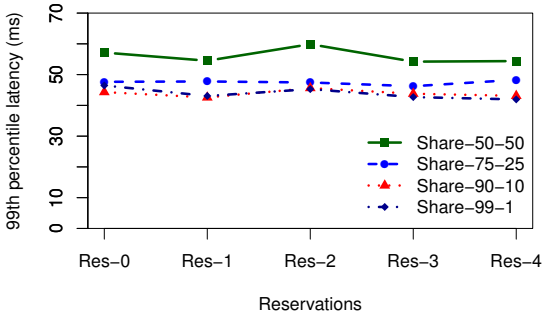


Figure 9: When the front-end is sending high throughput, the front-end queue at HDFS remains full and proportional share is an effective mechanism at reducing latency. Reservations are not as effective for this type of workload.

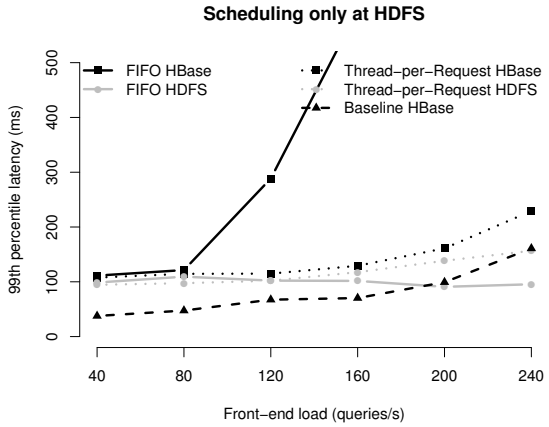


Figure 10: A thread-per-request policy at HBase leads to increased latency at both HBase and HDFS. FIFO has unbounded HBase queue time as load increases, but HDFS continues to provide good latency. This can be improved upon by non-FIFO scheduling policies.

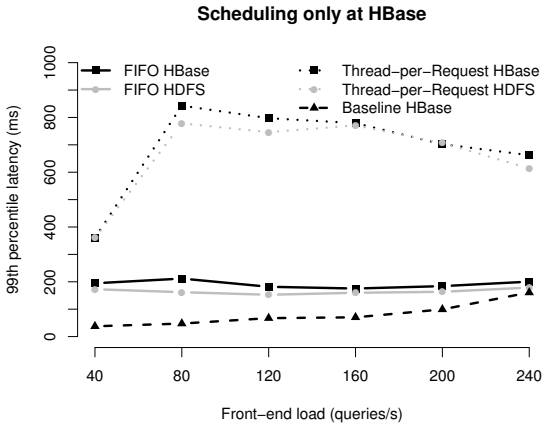


Figure 11: A thread-per-request policy at HDFS displays greatly increased latency compared to a basic FIFO policy with chunked request sizes.

6.1 Proportional Shares and Reservations

In this series of experiments, we evaluated the effectiveness of proportional shares and reservations at controlling latency at a single resource. HDFS was set to use a soft-limit of 6 handler threads in its thread pool. HBase was set to effectively a thread-per-request model by fixing the size of the HBase thread pool to be greater than the number of client threads, such that no queuing happened at HBase. This made HDFS the sole point of scheduling within the system. We evaluated these mechanisms by fixing different proportional share and reservation values at HDFS, and tested at different levels of constant front-end load contending with an unthrottled batch load.

Figure 8 plots the 99th percentile latency of the front-end client when varying the HDFS reservation and amount of front-end load. At low front-end load ($x = 50$), front-end latency suffers without a reservation. This is because the proportional share scheduler is work conserving and will choose to run a batch request when the front-end queue is empty, which happens frequently with low front-end load. However, as reservation is increased, front-end latency progressively improves. This is because reserved handlers will idle rather than running a batch request, allowing newly arriving front-end requests to be handled sooner. At high front-end load, the latency values converge since both batch and front-end queues remain full and front-end latency is now dominated by waiting for other front-end requests, not batch requests.

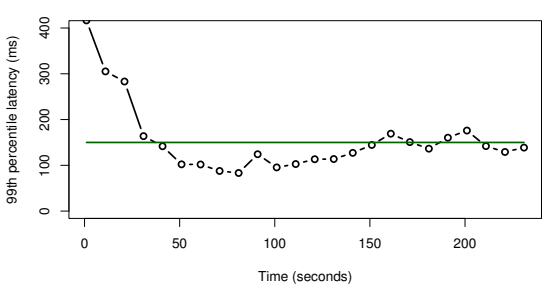


Figure 12: Cake quickly brings latency below the specified SLO of 150ms after approximately 40 seconds. Afterwards, Cake conservatively decreases the front-end's allocation when it sees the SLO is safely being met until it converges at $t = 150$.

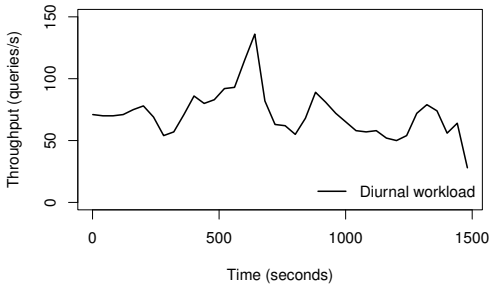


Figure 13: Front-end get throughput varies according to the diurnal pattern from a one-day trace of a real web-serving workload. The workload shows a 3x peak-to-trough difference.

Figure 9 shows the effectiveness of shares in these high-load scenarios. We fixed front-end load at 200 queries per second and ran different combinations of share and reservations. First, we again observe that reservations do not have a significant effect on latency at high load. However, proportional share does have a measurable effect: changing share from an even 50-50 to a more lopsided 90-10 improved the front-end's latency by approximately 20%.

In short, both shares and reservations have useful properties depending on the workload. Reservations are necessary at low load when arrival patterns are bursty and queues often become empty. Proportional share becomes effective at high load and allows for more granular adjustments in performance. Cake leverages both for different workload scenarios, allocating reservations when necessary but otherwise using proportional share.

6.2 Single vs. Multi-resource Scheduling

Next, we ran experiments to demonstrate the importance of coordinated, multi-resource scheduling. We tested the extent of the scheduling ability of the two possible single-resource configurations: (1) only scheduling at HDFS and (2) only scheduling at HBase. We again used YCSB to simulate front-end and batch clients. Front-end load was varied by adding an additional thread for each additional 10 queries per second increase.

We measured the 99$^{\text{th}}$ percentile of the front-end client in two places: the overall latency of requests to HBase (which includes time spent waiting for HDFS), and the latency of requests made to HDFS on behalf of the front-end. We compare these against the baseline total latency of a front-end client running without a contending batch workload.

First, we examine scheduling just at HDFS, without HBase scheduling. HDFS was configured to use a strong scheduling policy favoring the front-end client, with a reservation of 5 and share of 90-10 for the front-end. HBase was configured to not schedule, using either a thread-per-request model or a fixed-size thread pool with a single FIFO request queue.

The results are shown in Figure 10. With thread-per-request, latency increases when front-end load goes beyond approximately 160 queries/s. This is indicative of increased CPU contention within HBase when running many concurrent threads. With FIFO, latency increases greatly as additional front-end load is added. This is due to head-of-line blocking in HBase; without separate queues and differentiated scheduling, front-end requests must wait behind batch requests to run.

Next, we examine scheduling just at HBase, without HDFS scheduling. HBase was configured to use a proportional share of 99-1 strongly favoring the front-end client. HDFS was configured either with a thread-per-request model or FIFO with a fixed size thread pool and splitting of large requests.

The results are shown in Figure 11. With thread-per-request, latency increases greatly since the number of outstanding requests to disk is not limited. Interestingly, performance improves slightly at higher front-end load because of a higher proportion of front-end requests arriving at HDFS, but is still poor in absolute terms. With FIFO and chunking of large requests, latency is fairly constant at around 200ms, but still much higher than baseline, or the 100ms achieved by scheduling at just HDFS. This is because, even with chunking, front-end and batch requests still wait in a single queue.

Together, these results demonstrate that neither single-resource scheduling option can provide the latency control we require. Thread-per-request models fail to limit the number of outstanding requests, leading to severe resource contention and high latency. Using FIFO with a thread pool reduces resource contention, but is still unable to effectively control latency because all requests still wait in a single queue.

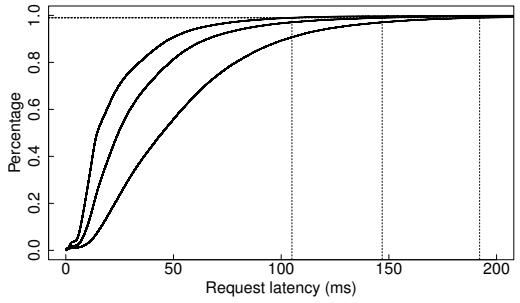


Figure 14: CDF of front-end latency for 100ms, 150ms, and 200ms 99th percentile latency SLOs for the diurnal workload. The 150ms and 200ms SLOs were met. Cake barely misses the 100ms SLO, with the actual 99th percentile latency at 105ms.

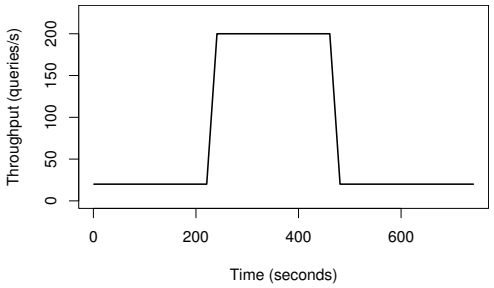


Figure 15: Front-end get throughput starts out at 20 queries/s, spikes up to 200 queries/s, then falls back to 20 queries/s. This spike is a 10x difference in peak-to-trough load, and represents a difficult case for scheduling.

6.3 Convergence Time

In this experiment, we evaluate the ability of the second-level Cake scheduler to converge to a specified latency SLO. We used YCSB to simulate a fixed front-end workload of 100 queries per second running alongside an unthrottled batch client. At the start of the experiment, both clients had 50% share and no reservation at each resource. A 150ms 99th percentile SLO was placed on the front-end workload.

Figure 12 depicts the front-end client's 99th percentile latency over time. We observe that latency is initially around 400ms, then decreases rapidly due to the changes made by the second-level scheduler. The 99th percentile latency falls below the SLO after approximately 40 seconds, or four second-level scheduler intervals. When Cake detects that the front-end is easily meeting its SLO, the front-end's allocation is gradually reduced as described in §5.3. Latency gradually increases over time, until it eventually converges to the SLO value at approximately $t = 150$. This convergence time could potentially be reduced by decreasing the scheduling interval from 10 seconds, but appears sufficient.

6.4 Diurnal Workload

To evaluate Cake's ability to adapt to changing workload patterns, we used an hourly trace from a user-facing web-serving workload from one of our industrial partners (Figure 13). The trace was sped up to run the full one-day trace in a 24 minute period, resulting in a highly dynamic pattern. Note that there is roughly a 3x difference in peak-to-trough load, with the front-end load peaking at around 140 queries/s. We used YCSB to generate load in this experiment, with the front-end client running the diurnal trace alongside a batch client with unthrottled throughput.

We ran this experiment with three different 99th percentile latency SLOs to illustrate the latency vs. throughput trade-off. Figure 14 depicts the CDF request latency of each run over the entire interval. We see that Cake's empirical 99th percentile performance met the target for the 150ms and 200ms SLOs. Cake narrowly misses meeting the 100ms SLO target, providing an actual 99th percentile latency of 105ms. This is somewhat expected given that the 100ms SLO is only 2x the baseline 99th percentile latency of the system. Even minor degrees of queuing greatly affect request latency and leads to SLO violations. Additionally, the compressed diurnal workload is changing quite rapidly. This means latency can suffer while the second-level scheduler converges to a new allocation. This could be ameliorated by looking for additional ways to drive down latency in the first-level schedulers and improve convergence time of the second-level scheduler. However, in light of the generally good SLO compliance of the system, we did not further pursue these changes.

Table 2 shows the batch throughput for the same three diurnal experiments. As expected, batch throughput increases significantly when the front-end SLO is weakened from 100ms to 150ms. However, throughput does not increase when the SLO is further weakened from 150ms to 200ms. This is for a number of reasons. First, the attained throughput comes close to the maximum scan throughput of the system (approximately 50 queries/s). Second, the rapid changes in front-end load mean the scheduler is forced to take a conservative approach and never converges to the best allocation.

Figure 16 illustrates the allocation decisions made by the second-level scheduler during the 100ms run. We see that the SLO compliance-based phase gradually steps the front-end's reservation up to 5, at which point the SLO is met. Note that the scheduler decreases the front-end's share when normalized performance is > 1.2, which allows additional batch requests to run. The queue occupancy-based phase moves HBase share in lockstep with HDFS share, and makes small adjustments in HBase reservation when queue occupancy at the two becomes unbalanced.

Front-end SLO	Batch throughput
100ms	24.6 queries/s
150ms	41.2 queries/s
200ms	41.2 queries/s

Table 2: Average batch throughput for the diurnal workload.

Front-end SLO	Batch throughput
100ms	22.9 queries/s
150ms	38.4 queries/s
200ms	45.0 queries/s

Table 3: Average batch throughput for the spike workload.

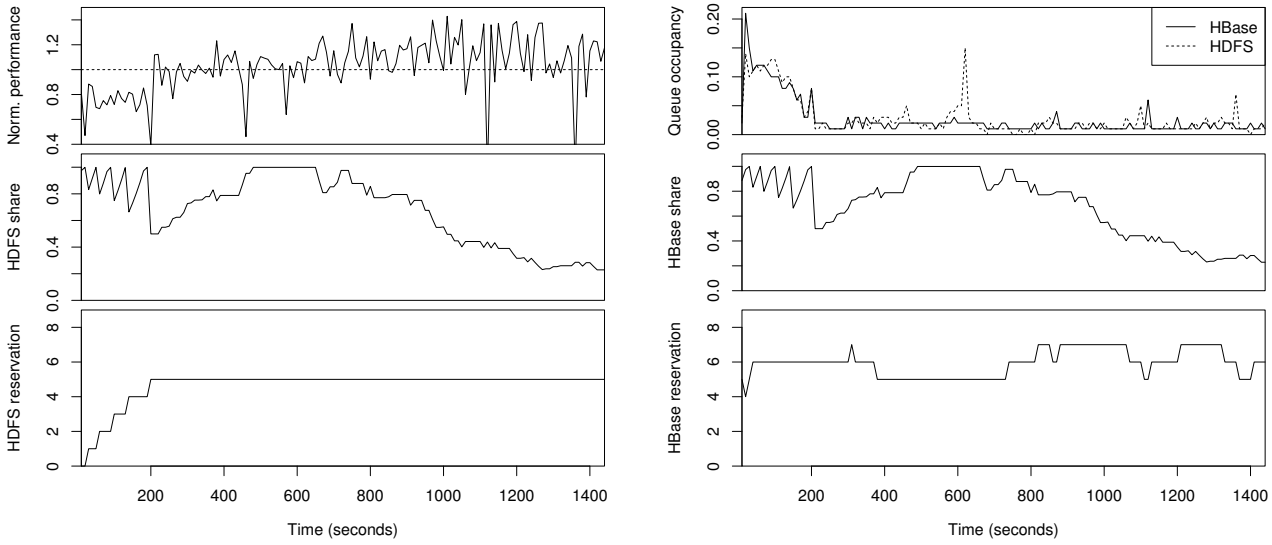


Figure 16: Second-level scheduler actions at HBase and HDFS for the diurnal workload with a 100ms SLO. Figure 16(a) shows that the SLO compliance-based algorithm keeps normalized performance around the set SLO value. Figure 16(b) shows that the queue occupancy-based algorithm adjusts reservations to keep HBase and HDFS in balance.

6.5 Spike Workload

Next, we tested Cake's ability to deal with sudden traffic spikes. The front-end YCSB client was configured to run the synthetic spike workload depicted in Figure 15. In this workload, the front-end's throughput starts out at 20 queries/s for four minutes, spikes up to 200 queries/s for four minutes, and then returns to 20 queries/s for another four minutes. During the spike, the front-end is sending close to the maximum attainable get throughput of the system.

The CDF of front-end request latency for 100ms, 150ms, and 200ms SLOs is shown in Figure 17. We see that Cake successfully enforces the 150ms and 200ms SLOs, indicating that it is able to adapt its scheduling even when dealing with a rapid change in the workload. The 100ms SLO is barely missed, with the measured 99th percentile latency at 107ms. Similar to the diurnal workload results in §6.4, this is caused by the convergence time of the second-level scheduler.

Average batch throughput for the interval shows a clear trade-off between the three SLOs. Since the interval is dominated by periods of low load, the 200ms SLO achieves higher batch throughput than in the diurnal case. The 100ms and 150ms experiments have similar throughput to the diurnal workload, slightly lessened due to the extreme nature of the spike.

6.6 Latency Throughput Trade-off

Cake enables its operators to trade-off between the latency SLO of a front-end client and the throughput of a batch client. To quantify the nature of this trade-off, we measured the throughput obtained by the batch client as we varied the latency SLO of the front-end client with a constant workload.

Figure 18 shows that as the latency SLO is relaxed from 80ms to 200ms, the batch throughput obtained doubles from around 20 queries/s to 40 queries/s. However, the maximum throughput achieved by Cake is only 67% of the baseline of a batch client running in isolation. This is because we chose to conservatively soft-limit the number of HDFS handlers at 6. This low number allows Cake to satisfy a wide range of latency SLOs, but means that it fails to fully saturate disk queues when trying to achieve high throughput. Experiments run with a higher soft-limit show that Cake can achieve up to 84% of the baseline throughout with a higher latency SLO of 300ms.

The latency vs. throughput trade-off is a fundamental property of rotational storage media. Achieving low latency requires short queues on the device, since queue time is a large contributor to high-percentile latency. Conversely, achieving high throughput requires long queues on

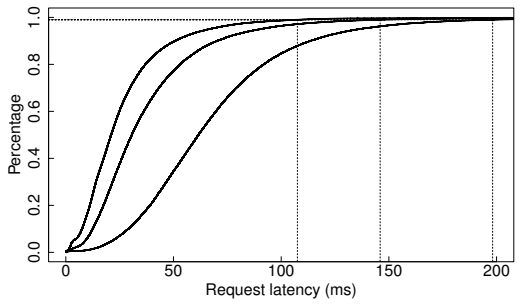


Figure 17: CDF of front-end latency for 100ms, 150ms, and 200ms 99th percentile latency SLOs for the spike workload. Again, Cake satisfies the 150ms and 200ms SLOs, and the 99th percentile latency for the 100ms SLO is 107ms

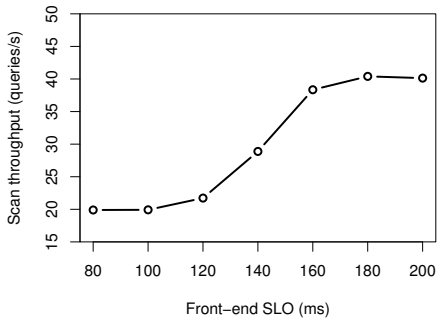


Figure 18: Batch throughput obtained increases from around 20 qps to 40 qps as the latency SLO for the front-end is relaxed from 80ms to 200ms.

Scenario	Time (seconds)
Baseline analysis	886.27±35.22
Estimated copy-then-process	1722±70.45
Consolidated analysis	1030±70.07

Table 4: We measured the time taken to analyze a set of 20 tables using MapReduce under different scenarios. We compare the baseline of MapReduce running by itself against MapReduce running concurrently with a front-end client with a 99th percentile latency SLO of 100ms. The total analysis time of the consolidated workload is only 16% greater than the baseline.

the device since it maximizes utilization and allows the OS and disk controller to more effectively merge and reorder requests. This trade-off is also partially defined by differences in random and sequential throughput.

6.7 Quantifying Benefits of Consolidation

To quantify the benefits of consolidating separate front-end and batch storage clusters, we simulated running an analytics report generation cycle on a 20-node Cake cluster on EC2. We created 20 tables in HBase containing a total of 386 GB of data. Front-end load was generated by an additional 20 nodes running the same diurnal trace from §6.4. We set a 99th percentile latency SLO of 100ms for the front-end. To generate our simulated report, we ran a set of MapReduce wordcount jobs which scanned over all the data in the tables. We ended each trial of the experiment when the last MapReduce job completed.

We compared report generation time in two scenarios: a consolidated front-end and batch cluster with Cake, and a traditional "copy-then-process" analytics cycle where the data needs to be copied between separate front-end and batch clusters before being analyzed. We estimated the time it would take a copy-then-process analytics cycle by doubling the baseline time it took to generate the report on a cluster running only the batch workload. The results are shown in Table 4. Each experiment was run five times, and we list the mean and standard deviation for each.

We see that Cake is very effective at reducing the analytics time through consolidation. Report generation takes 16% longer than the baseline comparison against a batch-only cluster, but runs in 60% of the estimated copy-then-process time. Averaged across all five consolidated runs, 99.5% of front-end requests met the 100ms 99th percentile latency SLO. In this scenario, provisioning costs can also be reduced by up to 50% since Cake allows separate front-end and batch clusters to be consolidated into a single cluster of half the total size, while still meeting existing latency and throughput performance requirements.

These results indicate that consolidation has increased benefits for a cluster than for a single machine. This is because actual batch loads do not have a uniform access pattern across the cluster. Individual nodes can experience significant periods of low batch load, during which they can provide excellent latency performance. This is especially true towards the end of the MapReduce job, when straggling tasks mean that only a small fraction of nodes are serving batch requests. This skew is caused by interplay between how data is distributed across HBase nodes and MapReduce task scheduling. One direction of future work is analyzing real-world MapReduce workload traces to better understand load skew and per-node utilization in batch clusters, and how this affects multiplexing batch and front-end workloads.

7. FUTURE WORK

SLO admission control. SLO admission control requires modeling the effect of composing multiple, heterogeneous client workloads on system performance based on the current level of hardware provisioning. This is complicated by how hard drives suffer from non-linearities performance when random and sequential workloads are combined. While Cake's existing scheduling mechanisms provide SLO enforcement, we also plan to investigate admission control techniques that can help Cake provide stronger guarantees on performance. This also involves being able to express how each client's performance should degrade when the system is overloaded.

Influence of DRAM and SSDs Industry trends indicate that solid state devices (SSDs) are emerging as a new tier in the datacenter storage hierarchy. Coupled with the use of large DRAM caches [15] in the datacenter, storage systems can use a combination of memory, SSDs and hard disks to satisfy user requests. We are interested in the ability of SSDs and DRAM to overcome the latency vs. throughput trade-off inherent in rotational media.

Composable application-level SLOs. For Cake, we chose to extend HBase, which allowed us to support a large class of front-end applications as well as MapReduce-style batch analytics. However, HBase is far from a common storage abstraction for the datacenter. Applications may desire direct disk access, an SQL interface, or some other storage API, yet still desire appropriate application-level performance guarantees. We believe that application-level performance guarantees can be built by composing guarantees provided by lower-level systems, allowing applications interacting at different levels of abstraction to all benefit from application-level end-to-end SLOs.

Automatic parameter tuning. Operating Cake involves choosing a number of system parameters, which can be dependent on the underlying hardware, workload, and performance requirements. This parameter space could potentially be reduced through more systematic and automatic means, making the system both more robust and simpler to configure.

Generalization to multiple SLOs. Cake currently only supports enforcement of a single client SLO. While this is sufficient for a large number of realistic deployment scenarios, Cake does not currently handle the case of multiple, potentially contending, latency and throughput SLOs from an array of clients. We feel that the core scheduling principles and ideas in Cake can be extended to the general case, but there are challenges involved from the consolidation of a large number of client workloads.

8. CONCLUSION

In conclusion, we have presented Cake, a coordinated, multi-resource scheduling framework for shared storage systems. Cake coordinates resource allocation across multiple software layers, and allows application programmers to specify their high-level SLOs directly to the storage system. Cake allows consolidation of latency-sensitive and throughput-oriented workloads while ensuring that 99th percentile latency SLOs of front-end clients are met. We evaluated Cake with a number of workloads derived from real-world traces, and show that Cake allows users to flexibly move within the storage latency vs. throughput trade-off by choosing different high-level SLOs. Furthermore, we show that consolidation with Cake has significant performance and economic improvements over copy-then-process analytics cycles, showing that we can reduce completion times by 40% while also reducing provisioning costs by up to 50%.

9. ACKNOWLEDGEMENTS

We would like to thank our colleagues in the AMP Lab at UC Berkeley and our three anonymous SOCC reviewers for their helpful comments and suggestions.

This research is supported in part by NSF CISE Expeditions award CCF-1139158, gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

10. REFERENCES

[1] Hadoop distributed file system. `http://hadoop.apache.org/hdfs`.

[2] Hbase. `http://hbase.apache.org`.

[3] The Apache Cassandra Project. `http://cassandra.apache.org/`.

[4] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg. Providing a cloud network infrastructure on a supercomputer. In *HPDC '10*, Chicago, IL.

[5] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-Independent Storage for Social Computing Applications. In *CIDR*, Asilomar, CA, 2009.

[6] L. A. Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. In *ISCA '11*, San Jose, USA.

[7] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes Realtime at Facebook. In *SIGMOD '11*, Athens, Greece.

[8] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, Oct. 1995.

[9] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *IEEE International Conference on Multimedia Computing an Systems (ICMCS '99)*, pages 400–405, 1999.

[10] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *22th International Symposium on Reliable Distributed Systems (SRDS03)*, pages 109–118, 2003.

[11] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4, 2008.

[12] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB 2008*, Auckland, NZ.

[13] J. Dean and L. Barroso. `http://research.google.com/people/jeff/latency.html`, March 26, 2012.

[14] A. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys '12*, Bern, Switzerland.

[15] G. Ananthanarayanan, A. Ghodsi, A. Wang, S. Shenker, I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI '12*, San Jose, CA, 2012.

[16] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson. Statistics-driven workload modeling for the cloud. In *ICDE '10*, Long Beach, CA.

[17] G. Ganger, J. Strunk, and A. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, 2003.

[18] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queuing for packet processing. In *SIGCOMM'12*, Helsinki, Finland, 2012.

[19] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI'11*, Boston, MA, 2011.

[20] P. Goyal, H. Vin, and H. Cheng. Start-time fair queueing: A Scheduling Algorithm for Integrated services Packet Switching Networks. *Networking, IEEE/ACM Transactions on*, 5(5):690–704, Oct 1997.

[21] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST'09*, pages 85–98, San Jose, CA, 2009.

[22] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO load balancing across storage devices. In *FAST '10*, San Jose.

[23] A. Gulati, A. Merchant, and P. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *OSDI '10*, Vancouver, Canada.

[24] J. Hamilton. The cost of latency. `http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx`.

[25] Y. Izrailevsky. NoSQL at Netflix. `http://techblog.netflix.com/2011/01/nosql-at-netflix.html`.

[26] V. Jacobson. Congestion avoidance and control. *SIGCOMM Computer Communication Review*, 25(1):157–187, Jan. 1995.

[27] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST'03*, pages 131–144, San Francisco, CA, 2003.

[28] H. V. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. scc: Cluster Storage Provisining Informed by Application Characteristics and SLAs . In *FAST'12*, San Jose, USA.

[29] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *ICAC '11*, pages 245–254, Karlsruhe, Germany.

[30] M. P. Mesnier and J. B. Akers. Differentiated storage services. *SIGOPS Operating Systems Review*, 45(1):45–53, Feb. 2011.

[31] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07*, pages 289–302, Lisbon, Portugal, 2007.

[32] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search, 2009.

[33] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS 1997*, pages 44–55.

[34] G. Soundararajan and C. Amza. Towards End-to-End Quality of Service: Controlling I/O Interference in Shared Storage Servers. In *Middleware 2008*, volume 5346, pages 287–305.

[35] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST '09*, pages 71–84, San Francisco, California, 2009.

[36] B. Trushkowsky, P. Bodık, A. Fox, M. Franklin, M. Jordan, and D. Patterson. The SCADS Director: Scaling a distributed storage system under stringent performance requirements. In *FAST 2011*, pages 163–176.

[37] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *FAST '07*, San Jose, CA, 2007.

[38] C. A. Waldspurger. Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. Technical Report MIT-LCS-TR-667, MIT, Laboratory for Computer Science, 1995.

[39] A. Wang, S. Venkataraman, S. Alspaugh, I. Stoica, and R. Katz. Sweet storage SLOs with Frosting. In *HotCloud 2012*, Boston, MA.

[40] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *ACM SIGCOMM 2011*, pages 50–61.

[41] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM 2012*, pages 139–150, Helsinki, Finland.