# Towards Practical Serverless Analytics

*Qifan Pu*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-105
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-105.html

June 25, 2019

# Towards Practical Serverless Analytics

by

Qifan Pu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joeseph Hellerstein
Professor Fernando Perez

Spring 2019

Towards Practical Serverless Analytics

**Abstract**

Towards Practical Serverless Analytics

by

Qifan Pu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Distributed computing remains inaccessible to a large number of users, in spite of many open source platforms and extensive commercial offerings. Even though many distributed computation frameworks have moved into the cloud, many users are still left to struggle with complex cluster management and configuration tools there.

In this thesis, we argue that cloud stateless functions represent a viable platform for these users, eliminating cluster management overhead, fulfilling the promise of elasticity. We first build a prototype system, PyWren, which runs on existing serverless function services, and show that this model is general enough to implement a number of distributed computing models, such as BSP. We then identify two main challenges to support truly practical and general analytics on a serverless platform. The first challenge is to facilitate communication-intensive operations, such as shuffle in the serverless setting. The second challenge is to provide an elastic cloud memory. In this thesis, we made progress on both challenges. For the first, we develop a system called Locus, that can automate shuffle operations by judiciously provisioning hybrid intermediate storage. For the second, we present an algorithm, FairRide, that achieves near-optimal memory cache efficiency in a multi-tenant setting.

To my family.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I am deeply indebted to my advisor, Ion Stoica, who tirelessly provided guidance throughout my time at Berkeley. Ion took me under his wing when I was little experienced with distributed systems and relatively new to research. Among all things, he taught me the value of simplicity in solving complex technical problems. The most frequent word Ion used during our meetings was the word "fundamentally". It was during those moments that I learned to pick up a researcher's primal instinct to always examine problems with a deeper lens.

I am also thankful to my committee members, Fernando Perez, Joseph Hellerstein, and my qualification committee Scott Shenker. All of them are my role models and have provided me with valuable insights and constructive feedbacks. I took much inspiration from their works in this thesis, which one can find many links.

None of the work is possible without help of my great collaborators. My first published paper at Berkeley is with Ganesh Ananthanarayanan and Srikanth Kandula, who were my mentors at Microsoft Research. I also enjoyed much advice from Ganesh over the years even after that collaboration. Haoyuan Li, Matei Zaharia and Ali Ghodsi collaborated with me on the FairRide project. All of them shaped my research methodology in their unique ways. Eric Jonas got me excited on the PyWren project, which forms the basis of this thesis. From day I started at the RISELab, I never stopped bugging Shivaram Venkataraman with questions. Shivaram was always patient to answer. When we collaborated on Locus, it was his encouragement and continuous help that pushed me through multiple paper rejections. I also owe tremendous to Anand Padmanabha Iyer, Allan Pang, Sameer Agarwal, Kai Zeng, Reynold Xin, Peter Bodik, Peter Boncz, Aditya Akella, Sylvia Ratnasamy for putting up with me.

I would also like to thank my mentors before graduate school, Wenjun Hu, Shyam Gollakota and Arvind Krishnamurthy. They opened the door of research for me, and were generous enough to teach me from scratch.

Beyond direct collaborators on research projects, many colleagues and friends at Berkeley contributed to my graduate study and have made the journey a wonderful experience. Michael Chang, Kaifei Chen, Mosharaf Chowdhury, Dan Crankshaw, Ankur Dave, Biye Jiang, Anurag Khandelwal, Gautam Kumar, Chang Lan, Richard Liaw, Radhika Mittal, Kay Ousterhout, Aurojit Panda, Johann Schleier-Smith, Colin Scott, Vaishaal Shankar, Justine Sherry, Vikram Sreekanti, Liwen Sun, Alexey Tumanov, Amin Tootoonchian, Di Wang, Yifan Wu, Stephanie Wang, Xin Wang, Neeraja Yadwadkar, Zongheng Yang, Zhao Zhang, Ben Zhang, Wenting Zheng, David Zhu... I am also grateful to the RISELab administrative staff, Kattt Atchley, Jon Kuroda, Boban Zarkovich, Carlyn Chinen and Shane Knapp, who have made the lab like home.

The place I will miss most at Berkeley is the Berkeley Art Museum and Pacific Film Archive. Thanks to my film buddies, Helen Jiang and Emerson Lin, I have watched many films at BAMPFA over my PhD years, without which I might have graduated earlier. Though I have no regret.

Last but not least, this dissertation is dedicated to my father and my mother, for their unwavering love.

# Chapter 1

# Introduction

## 1.1 The Promise of Cloud

The past decade has seen the widespread adoption of cloud computing infrastructure where users launch virtual machines on demand to deploy services on a provisioned cluster. As cloud computing continues to evolve towards more elasticity, there is a shift to using *serverless* computing, where storage and compute is separated for both resource provisioning and billing. This trend was started by services like Google BigQuery [16], and AWS Glue [44] that provide cluster-free data warehouse analytics, followed by services like Amazon Athena[10] that allow users to perform interactive queries against a remote object storage without provisioning a compute cluster. While the aforementioned services mostly focus on providing SQL-like analytics, to meet the growing demand, all major cloud providers now offer "general" serverless computing platforms, such as AWS Lambda, Google Cloud Functions, Azure Functions and IBM OpenWhisk. In these platforms short-lived user-defined functions are scheduled and executed in the cloud. Compared to virtual machines, this model provides more fine-grained elasticity with sub-second start-up times, so that workload requirements can be dynamically matched with continuous scaling.

Fine-grained elasticity in serverless platforms is naturally useful for on-demand applications like creating image thumbnails [36] or processing streaming events [61]. However, we observe such elasticity also plays an important role for data analytics workloads. Consider for example an ad-hoc data analysis job exemplified by say TPC-DS query 95 [86] (See Section 3.4 for more details). This query consists of eight stages and the amount of input data at each stage varies from 0.8MB to 66GB. With a cluster of virtual machines users would need to size the cluster to handle the largest stage leaving resources idle during other stages. Using a serverless platform can improve resource utilization as resources can be immediately released after use.

We argue that a *serverless* execution model with *stateless* functions can enable radically-simpler, fundamentally elastic, and more user-friendly distributed data processing systems. In this model, we have one simple primitive: users submit functions that are executed in a remote container; the functions are stateless as all the state for the function, including input, output is accessed from shared remote storage. Surprisingly, we find that the performance degradation from using such an approach is negligible for many workloads and thus, our simple primitive is in fact general enough to implement a number of higher-level data processing abstractions, including MapReduce and parameter servers.

## 1.2 Scaling Functions with Serverless

As a first step, we describe a prototype system, PyWren, developed in Python with AWS Lambda. By employing only stateless functions, PyWren helps users avoid the significant developer and management overhead that has until now been a necessary prerequisite. The complexity of state management can instead be captured by a global scheduler and fast remote storage. With PyWren, we seek to understand the trade-offs of using stateless functions for large scale data analytics and specifically what is the impact of solely using

remote storage for inputs and outputs. We find that we can achieve around 30-40 MB/s write and read performance per core to a remote bulk object store (S3), matching the per-core performance of a single local SSD on typical EC2 nodes. Further we find that this scales to 60-80 GB/s to S3 across 2800 simultaneous functions, showing that existing remote storage systems may not be a significant bottleneck.

Using this as a building block we implement image processing pipelines where we extract per-image features during a map phase via unmodified Python code. We also show how we can implement BSP-style applications on PyWren and that a word count job on 83M items is only 17% slower than PySpark running on dedicated servers. Shuffle-intensive workloads are also feasible as we show PyWren can sort 1TB data in 3.4 minutes. However, we do identify two major challenges in making such serverless analytics systems more practical, which we discuss in the next sections.

## 1.3 Challenge 1: Communication in Serverless Analytics

Directly using a serverless platform for data analytics workloads could lead to extremely inefficient execution. For example we find that running the CloudSort benchmark [102] with 100TB of data on AWS Lambda, can be up to 500× slower (Section 3.1.3) when compared to running on a cluster of VMs. By breaking down the overheads we find that the main reason for the slowdown comes from slow data shuffle between asynchronous function invocations. As the ephemeral, stateless compute units lack any local storage, and as direct transfers between functions is not always feasible[1], intermediate data between stages needs to be persisted on shared storage systems like Amazon S3. The characteristics of the storage medium can have a significant impact on performance and cost. For example, a shuffle from 1000 map tasks to 1000 reduce tasks leads to 1M data blocks being created on the storage system. Therefore, throughput limits of object stores like Amazon S3 can lead to significant slow downs (Section 3.1.3).

Our key observation is that in addition to using elastic compute and object storage systems we can also provision *fast* memory-based resources in the cloud, such as in-memory Redis or Memcached clusters. While naively putting all data in fast storage is cost prohibitive, we can appropriately combine fast, but expensive storage with slower but cheaper storage, similar to the memory and disk hierarchy on a local machine, to achieve the best of both worlds: approach the performance of a pure in-memory execution at a significantly lower cost. However, achieving such a sweet spot is not trivial as it depends on a variety of configuration parameters, including storage type and size, degree of task parallelism, and the memory size of each serverless function. This is further exacerbated by the various performance limits imposed in a serverless environment (Section 3.1.4).

In Chapter 3 we propose Locus, a serverless analytics system that combines multiple storage types to achieve better performance and resource efficiency. In Locus, we build a performance model to aid users in selecting the appropriate storage mechanism, as well as the amount of fast storage and parallelism to use for map-reduce like jobs in serverless environments. Our model captures the performance and cost metrics of various cloud storage systems and we show how we can combine different storage systems to construct hybrid shuffle methods. Using simple micro-benchmarks, we model the performance variations of storage systems as other variables like serverless function memory and parallelism change.

We evaluate Locus on a number of analytics applications including TPC-DS, Daytona CloudSort and the Big Data Benchmark. We show that using fine-grained elasticity, Locus can reduce cluster time in terms of total core·seconds by up to 59% while being close to or beating Spark's query completion time by up to 2×. We also show that with a small amount of fast storage, for example, with fast storage just large enough to hold 5% of total shuffle data, Locus matches Apache Spark in running time on CloudSort benchmark and is within 13% of the cost of the winning entry in 2016. While we find Locus to be 2× slower when compared to Amazon Redshift, Locus is still a preferable choice to

[1]Cloud providers typically provide no guarantees on concurrent execution of workers.

Redshift since it requires no provisioning time (vs. minutes to setup a Redshift cluster) or knowing an optimal cluster size beforehand. Finally, we also show that our model is able to accurately predict shuffle performance and cost with an average error of 15.9% and 14.8%, respectively, which allows Locus to choose the most appropriate shuffle implementation and other configuration variables.

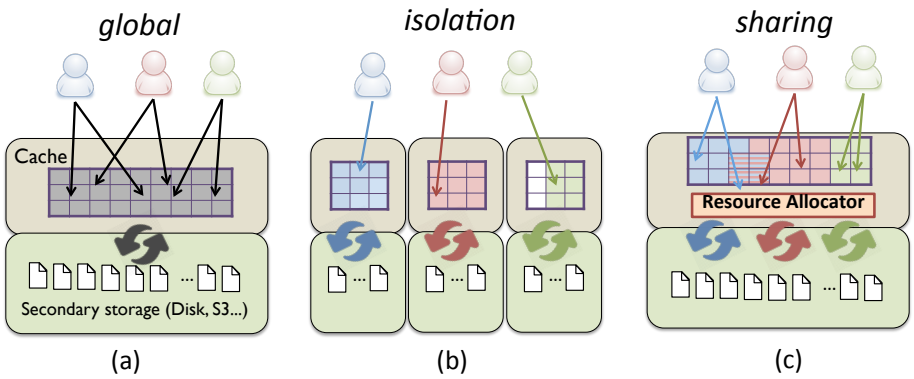



Figure 1.1: Different schemes. *Global*: single memory pool, agnostic of users or applications; *Isolation*: static allocations of memory among multiple users, possibly underutilization (blank cells), no sharing; *Sharing*: allowing dynamic allocations of memory among users, and one copy of shared files (stripe cells).

## 1.4 Challenge 2: Practical Elastic Memory

While Locus leverages elastic memory service in the cloud, those services are not provided in a "serverless" fashion, i.e., users need to provision machines in order to use them, e.g., Amazon ElastiCache. A truely serverless elastic memory service, just like serverless storage such as S3, should allow users to instantaneously acquire/release memory based on demand, and only be charged at usage. One step towards such service is to solving the problem of how multiple users can share memory in a fair, efficient manner. We study this problem in the *FairRide* project.

Unfortunately, we find that traditional caching policies do not provide a satisfactory answer to this problem. Most cache management algorithms (e.g., LRU, LFU) have focused on *global efficiency* of the cache (Figure 1.1a): they aim to maximize the overall hit rate. Regardless of being commonly used in today's cache systems for cloud serving (Redis [88], Memcached [71]) and big data storage (HDFS Caching [50]), this has two problems in a shared environment. First, users who read data at long intervals may gain little or no benefit from the cache, simply because their data is likely to be evicted out of the memory. Second, applications can also easily *abuse* such systems by making spurious accesses to increase their access rate. There is no incentive to dissuade users from doing this in a cloud environment, and moreover, such shifts in the cache allocation can happen even with non-malicious applications. We show later that a strategic user can outperform a non-strategic user by 2.9×, simply by making spurious accesses to her files.

The other common approach is to have *isolated caches* for each user (Figure 1.1b). This gives each user performance guarantees and there are many examples in practice, e.g., hypervisors that set up separate buffer caches for each of its guest VMs, web hosting platforms that launch a separate `memcached` instance to each tenant. However, providing

such performance guarantees comes at the cost of inefficient utilization of the cache.

This inefficiency is not only due to users not fully utilizing their allocated cache, but also because that a cached file can be accessed by multiple users at a time and isolating cache leads to multiple copies of such *shared files*. We find such *non-exclusive sharing* to be a defining aspect of cache allocation, while other resources are typically exclusively shared, e.g., a CPU time slice or a communication link can be only used by a single user at a time. In practice, there are a significant number of files shared across users in many workloads, e.g., we observe more than 30% files are shared by at least two users from a production HDFS log. Such sharing is likely to increase as more workloads move to multi-tenant environments.

In Chapter 4, we study how to share cache space between multiple users that access shared files. To frame the problem, we begin by identifying desirable properties that we'd like an allocation policy to have. Building on common properties used in sharing of CPU and network resources [42], we identify three such properties:

- *Isolation Guarantee*: no user should receive less cache space than she would have had if the cache space were statically and equally divided between all users (i.e., assuming $n$ users and equal shares, each one would get $1/n$ of the cache space). This also implies that the user's cache performance (e.g., cache miss ratio) should not be worse than isolation.

- *Strategy Proofness*: a user cannot improve her allocation or cache performance at the expense of other users by gaming the system, e.g., through spuriously accessing files.

- *Pareto Efficiency*: the system should be *efficient*, in that it is not possible to increase one user's cache allocation without lowering the allocation of some other user. This property captures operator's desire to achieve high utilization.

These properties are common features of allocation policies that apply to most resource sharing schemes, including CPU sharing via lottery or stride scheduling [112, 20, 106, 113], network link sharing via max-min fairness [68, 14, 30, 46, 99, 105], and even allocating multiple resources together for compute tasks [42]. Somewhat unexpectedly, there has been no equivalent policy for allocation of cache space that satisfies all three properties. As shown earlier, global sharing policies (Figure 1.1a) lack isolation-guarantee and strategy-proofness, while static isolation (Figure 1.1b) is not Pareto-efficient.

The first surprising result we find is that this deficiency is no accident: in fact, for sharing cache resources, *no policy can achieve all three properties*. Intuitively, this is because cached data can be shared across multiple users, allowing users to game the system by "free-riding" on files cached by others, or optimizing usage by caching popular files. This creates a strong trade-off between Pareto efficiency and strategy-proofness.

While no memory allocation policy can satisfy the three properties (Table 1.1), we show that there are policies that come close to achieving all three in practice. In particular, we

Table 1.1: Summary of various memory allocation policies against three desired properties.

| | Isolation Guarantee | Strategy-Proofness | Pareto-Efficiency |
|---|---|---|---|
| **global (e.g., LRU)** | ✗ | ✗ | ✓ |
| **max-min fairness** | ✓ | ✗ | ✓ |
| **FairRide** | ✓ | ✓ | near-optimal |
| `None Exist` | ✓ | ✓ | ✓ |

propose FairRide, a policy that provides both isolation-guarantee (so it always performs no worse than isolated caches) and strategy-proofness (so users are not incentivized to cheat), and comes within 4% of global efficiency in practice. FairRide does this by aligning each user's benefit-cost ratio with her private preference, through *probabilistic blocking* (Section 4.2.4), i.e., probabilistically disallowing a user from accessing a cached file if the file is not cached on behalf of the user. Our proof in Section 4.4 shows that *blocking* is required to achieve strategy-proofness, and that FairRide achieves the property with minimal blocking possible.

In practice, *probabilistic blocking* can be efficiently implemented using *expected delaying* (Section 4.3.1) in order to mitigate I/O overhead and to prevent even more sophisticated cheating models. We implemented FairRide on Tachyon [62], a memory-centric storage system, and evaluated the system using both cloud serving and big data workloads. Our evaluation shows that FairRide comes within 4% of global efficiency while preventing strategic users, meanwhile giving 2.6× more job run-time reduction over isolated caches. In a non-cooperative environment when users do cheat, FairRide outperforms max-min fairness by at least 27% in terms of efficiency. It is also worth noting that FairRide would support pluggable replacement policies as it still obeys each user's caching preferences, which allows users to choose different replacement policies (e.g., LRU, LFU) that best suit their workloads.

## 1.5 Dissertation Roadmap

The rest of the dissertation is organized as follows. Chapter 2 gives a background of serverless computing and presents PyWren, a prototype system which allows users to run elastic functions on the serverless infrastructure. While PyWren can be used to express a wide range of applications, we also identify bottlenecks, i.e., communication-intensive operations and storage backend.

Chapter 3 describes our efforts in addressing the first challenge, i.e., to support cost-efficient shuffles while maintaining performance compared to traditional approaches. In this chapter, we present Locus, a system that can judiciously combine different storage types for a given shuffle.

Chapter 4 starts to solve the problem of how to provide elastic memory in the cloud. Elastic memory is not available in today's cloud. One key reason is that sharing memory across multiple users is difficult. We study the problem and derive theoretical results on the impossibility of attaining all good properties that have been previously achieved by other resource types. In this chapter, we also present a new policy called FairRide.

Finally we conclude in Chapter 5.

## Chapter 2

# Simplifying Data Analytics with Serverless Functions

Distributed computing remains inaccessible to a large number of users, in spite of many open source platforms and extensive commercial offerings. While distributed computation frameworks have moved beyond a simple map-reduce model, many users are still left to struggle with complex cluster management and configuration tools, even for running simple embarrassingly parallel jobs. We argue that stateless functions represent a viable platform for these users, eliminating cluster management overhead, fulfilling the promise of elasticity. Furthermore, using our prototype implementation, PyWren, we show that this model is general enough to implement a number of distributed computing models, such as BSP, efficiently. Extrapolating from recent trends in network bandwidth and the advent of disaggregated storage, we suggest that stateless functions are a natural fit for data processing in future computing environments.

## 2.1 Is the Cloud Usable?

The advent of elastic computing has greatly simplified access to computing resources, as the complexity of management is now handled by cloud providers. Thus the complexity has now shifted to applications or programming frameworks. However most software, especially in scientific and analytics applications, is not written by computer scientists [53, 72], and it is many of these users who have been left out of the cloud revolution.

The layers of abstraction present in distributed data processing platforms are complex and difficult to correctly configure. For example, PySpark, arguably one of the easier to use platforms, runs on top of Spark [117] (written in Scala) which interoperates and is closely coupled with HDFS [101] (written in Java), Yarn [109] (Java again), and the JVM. The JVM in turn is generally run on virtualized Linux servers. Merely negotiating the memory limit interplay between the JVM heap and the host operating system is an art form [35, 108, 104]. These systems often promote “ease of use" by showing powerful functionality with a few lines of code, but this ease of use means little without mastering the configuration of the layers below.

In addition to the software configuration issues, cloud users are also immediately faced with tremendous planning and workload management before they even begin running a job. AWS offers 70 instances types across 14 geographical datacenters – all with subtly different pricing. This complexity is such that recent research has focused on algorithmic optimization of workload trade-offs [52, 110]. While several products such as Databricks and Qubole simplify cluster management, the users still need to explicitly start and terminate clusters, and pick the number and type of instances.

Finally, the vast majority of scientific workloads could take advantage of dynamic market-based pricing of servers, such as AWS spot instances – but computing spot instance pricing is challenging, and additionally most of the above-mentioned frameworks make it difficult to handle machine preemption. To avoid the risk of losing intermediate data, users must be careful to either regularly checkpoint their data or run the master and a certain number of workers on non-spot instances. This adds another layer of management complexity which makes elasticity hard to obtain in practice.

**What users want:** Our proposal in this chapter was motivated by a professor of computer graphics at UC Berkeley asking us “Why is there no cloud button?" He outlined how his students simply wish they could easily “push a button" and have their code – existing, optimized, single-machine code – running on the cloud. Thus, our fundamental goal here is to allow as many users as possible to take existing, legacy code and run it in parallel, exploiting elasticity. In an ideal world, users would simply be able to run their desired code across a large number of machines, bottlenecked only by serial performance. Executing 100 or 10000 five-minute jobs should take roughly five minutes, with minimal start-up and tear-down overhead.

Further, in our experience far more users are capable of writing reasonably-performant single-threaded code, using numerical linear algebra libraries (e.g., OpenBLAS, Intel’s

MKL), than writing complex distributed-systems code. Correspondingly the goal for these users is not to get the best parallel performance, but rather to get vastly better performance than available on their laptop or workstation while taking *minimal development time*.

For compute-bound workloads, it becomes more useful to parallelize across functions for many cases; to say sweep over a wide range of parameters (such as machine learning hyperparameter optimization) or try a large number of random initial seeds (Monte Carlo simulations of physical systems). In these cases, exposing function-level parallelism is more worthwhile than having complex interfaces for intra-function optimization. Therefore, a simple function interface that captures sufficient local state, performs computation remotely, and returns the result is more than adequate. For data-bound workloads, a large number of users would be served by a simpler version of the existing map-reduce framework where outputs can be easily persisted on object storage.

Thus, a number of compute-bound and data-bound workloads can be captured by having a simple abstraction that allows users to run arbitrary functions in the cloud without setting up and configuring servers/frameworks etc. We next discuss why such an abstraction is viable now and the components necessary for such a design.

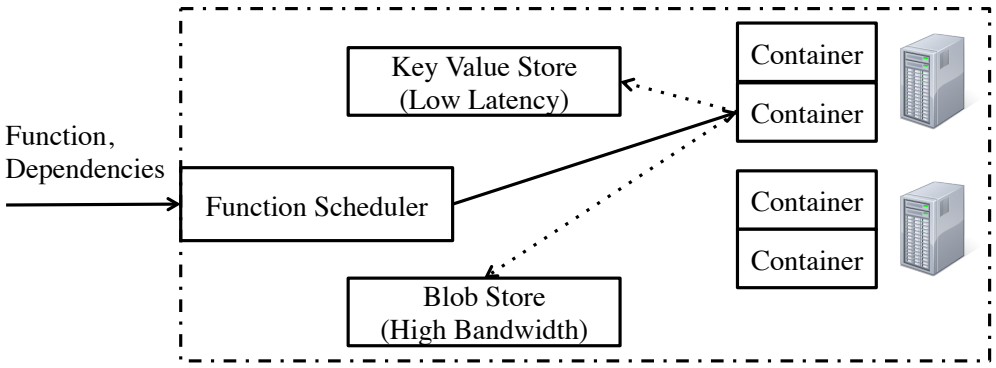



Figure 2.1: System architecture for stateless functions.

## 2.2 A Modest Proposal

Many of the problems with current cloud computing abstractions stem from the fact that they are designed for a server-oriented resource model. Having servers as the unit of abstraction ties together multiple resources like memory, CPU and network bandwidth. Further servers are also often long running and hence require DevOps support for maintenance. Our proposal is to instead use a serverless architecture with *stateless functions* as the unifying abstraction for data processing. Using stateless functions will simplify programming and deployment for end users. In this section we present the high level components for designing data processing systems on a serverless architecture. While other proposals [11] have looked at implementing data processing systems on serverless infrastructure, we propose a simple API that is tightly integrated with existing libraries and also study performance trade-offs of this approach by using our prototype implementation on a number of workloads.

### 2.2.1 Systems Components

The main components necessary for executing stateless functions include a low overhead execution runtime, a fast scheduler and high performance remote storage as shown in Figure 2.1. Users submit single-threaded functions to a global scheduler and while submitting the function they can also annotate the runtime dependencies required. Once the scheduler determines where a function is supposed to run, an appropriate container is created for the duration of execution. While the container maybe reused to improve performance none of the state created by the function will be retained across invocations. Thus, in such a model all the inputs to functions and all output from functions need to be persisted on remote storage and we include client libraries to access both high-throughput and low latency shared storage systems.

**Fault Tolerance**: Stateless functions allow simple fault tolerance semantics. When a function fails, we restart it (at possibly a different location) and execute on the same input. We only need atomic writes to remote storage for tracking which functions have succeeded. Assuming that functions are idempotent we obtain similar fault tolerance

Table 2.1: Comparison of single-machine write bandwidth to instance local SSD and remote storage in Amazon EC2. Remote storage is faster than single SSD on the standard `c3.8xlarge` instance and the storage-optimized `i2.8xlarge` instance.

| Storage Medium | Write Speed (MB/s) |
|---|---|
| SSD on `c3.8xlarge` | 208.73 |
| SSD on `i2.8xlarge` | 460.36 |
| 4 SSDs on `i2.8xlarge` | 1768.04 |
| S3 | 501.13 |

guarantees as existing systems.

**Simplicity**: As evidenced by our discussion above, our architecture is very simple and only consists of the minimum infrastructure required for executing functions. We do not include any distributed data structures or dataflow primitives in our design. We believe that this simplicity is necessary in order to make simple workloads like embarrassingly parallel jobs easy to use. More complex abstractions like dataflow or BSP can be implemented on top and we discuss this in Section 2.2.3.

**Why now?** The model described above is closely related to systems like Linda [21], Celias [48] and database trigger-based systems [87, 84]. While their ideas are used in work-stealing queues and shared file system, the specific programming model has not been widely adopted. We believe that this model is viable now given existing infrastructure and technology trends. While the developer has no control of where a stateless function runs (e.g., the developer cannot specify that a stateless function should run on the node storing the function's input), the benefits of colocating computation and data – a major design goal for prior systems like Hadoop, Spark and Dryad – have diminished.

Prior work has shown that hard disk locality does not provide significant performance benefits [37]. To see whether the recent datacenter migration from hard disks to SSDs has changed this conclusion, we benchmarked the I/O throughput of storing data on a local SSD of an AWS EC2 instance vs. storing data on S3. Our results, in Table 2.1, show that currently that writing to remote storage is faster than a single SSD but using multiple SSDs can yield better performance. However, technology trends [49, 98, 34] indicate that the gap between network bandwidth and storage I/O bandwidth is narrowing, and many recently published proposals for rack-scale computers feature disaggregated storage [55, 9] and even disaggregated memory [38]. All these trends suggest diminishing performance benefits from colocating compute with data in the future.

### 2.2.2 PyWren: A Prototype

We developed PyWren[1] to rapidly evaluate these ideas, seamlessly exposing a map primitive from Python on top of AWS Lambda. While Lambda was designed to run

[1] A wren is much smaller than a Condor

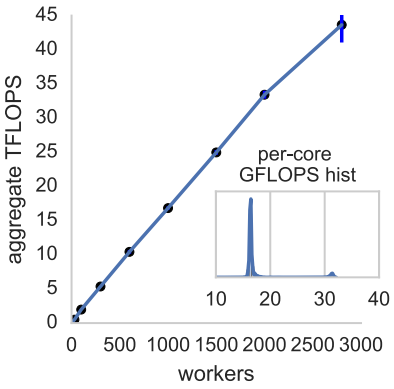



Figure 2.2: Running a matrix multiplication benchmark inside each worker, we see a linear scalability of FLOPs across 3000 workers.

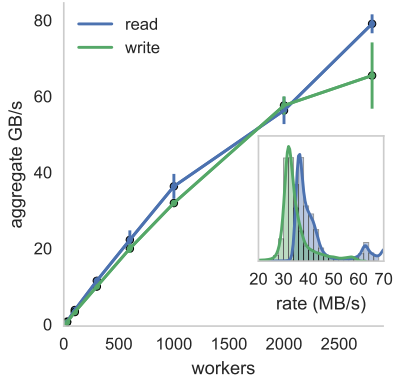



Figure 2.3: Remote storage on S3 linearly scales with each worker getting around 30 MB/s bandwidth (inset histogram).

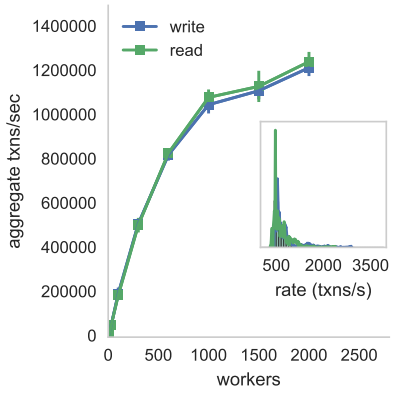



Figure 2.4: Remote key-value operations to Redis scales up to 1000 workers. Each worker gets around 700 synchronous transactions/sec.

event-driven microservices (such as resizing a single user-uploaded image) with a fixed funtion, by extracting new code from S3 during runtime we make each Lambda invocation run a different function. Currently AWS Lambda provides a very restricted containerized runtime with a maximum 300 seconds of execution time, 1.5 GB of RAM, 512 MB of local storage and no root access, but we believe these limits will be increased as AWS Lambda is used for more general purpose applications.

PyWren serializes a Python function using `cloudpickle` [26], capturing all relevant information as well as most modules that are not present in the server runtime[2]. This eliminates the majority of user overhead about deployment, packaging, and code versioning. We submit the serialized function along with each serialized datum by placing them into globally unique keys in S3, and then invoke a common Lambda function. On the server side, we invoke the relevant function on the relevant datum, both extracted from S3. The result of the function invocation is serialized and placed back into S3 at a pre-specified key, and job completion is signaled by the existence of this key. In this way, we are able to reuse one registered Lambda function to execute different user Python functions and mitigate the high latency for function registration, while executing functions that exceed Lambda's code size limit.

**Map for everyone**: As discussed in Section 2.1, many scientific and analytic workloads are embarrassingly parallel. The `map` primitive provided by PyWren makes addressing these use cases easy – serializing all local state necessary for computation, transparently

[2]While there are limitations in the serialization method (including an inability to transfer arbitrary Python C extensions), we find this can be overcome using libraries from package managers such as Anaconda.

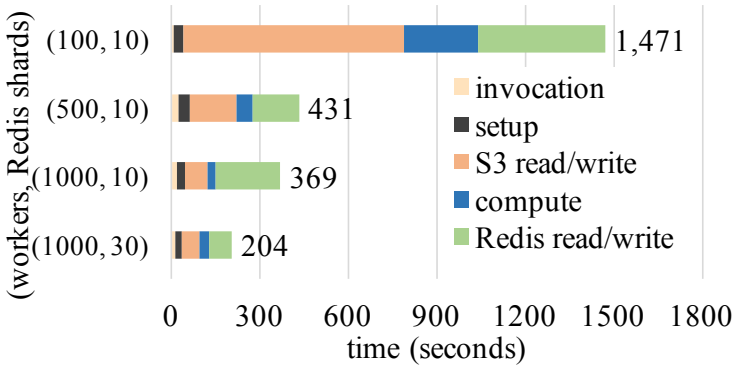



Figure 2.5: Performance breakdown for sorting 1TB data by how task time is spent on average.

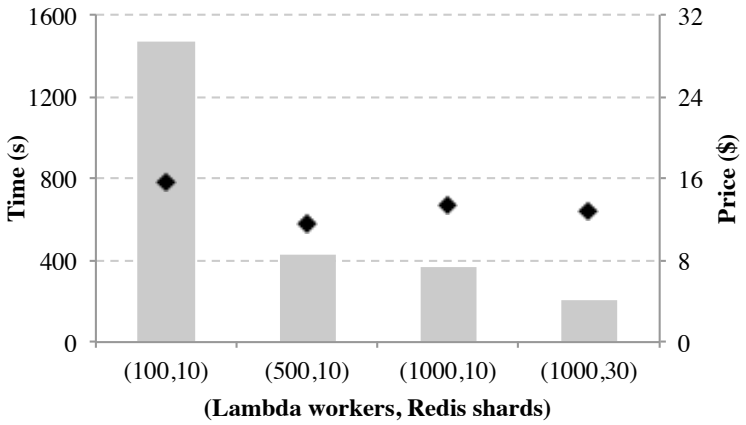



Figure 2.6: Prorated cost and performance for running 1TB sort benchmark while varying the number of Lambda workers and Redis shards.

invoking functions remotely and returning when complete. Calling `map` launches as many stateless functions as there are elements in the list that one is mapping over. An important aspect to note here is that this API mirrors the existing Python API for parallel processing and thus, unlike other serverless MapReduce frameworks [11], this integrates easily with existing libraries for data processing and visualization.

**Microbenchmarks**: Using PyWren we ran a number of benchmarks(Figures 2.2,2.3,2.4) to determine the impact of solely using remote storage for IO, and how this scales with worker count. In terms of compute, we ran a matrix multiply kernel within each Lambda and find that we get 18 GFLOPS per core and that this unsurprisingly scales to more than 40 TFLOPS while using 2800 workers. To measure remote I/O throughput we benchmarked the read, write bandwidth to S3 and our benchmarks show that we can get on average 30 MB/s write and 40 MB/s read per Lambda and that this also scales to more than 60 GB/s write and 80 GB/s read. Assuming that 16 such Lambdas are as powerful as a single

server, we find that the performance from Lambda matches the S3 performance shown in Table 2.1. To measure the overheads for small updates, we also benchmarked 128-byte synchronous put/gets to two `c3.8xlarge` instances running in-memory Redis. We match the performance reported in prior benchmarks [90] and get less than 1ms latency up to 1000 workers.

**Applications**: In our research group we have had students use PyWren for applications as diverse as computational imaging, scientific instrument design, solar physics, and object recognition. Working with heliphysicists at NASA's Solar Dynamics Observatory, we have used PyWren for extracting relevant features across 16TB of solar imaging data for solar flare prediction. Working with applied physics colleagues, we have used PyWren to design novel types of microscope point-spread functions for 3d superresolution microscopy. This necessitates rapid and repeat evaluation of a complex physics-based optical model inside an inner loop.

### 2.2.3 Generality for the Rest of Us?

While the map primitive in PyWren covers a number of applications, it prohibits any coordination among the various tasks. We next look at how stateless functions along with high performance storage can also be used as a flexible building block to develop more complex abstractions. We next discuss how the high bandwidth and throughput discussed in the previous section, can be used to support a number of machine learning and scientific applications.

**Map + monolithic Reduce** The first abstraction we consider is one where output from all the map operations is collected on to one machine (similar to `gather` in HPC literature) for further processing. We find this pattern covers a number of classical machine learning workloads which consist of a featurization (or ETL) stage that converts large input data into features and then a learning stage where the model is built using SVMs or linear classifiers. In such workloads, the featurization requires parallel processing but the generated features are often small and fit on a single large machine [18]. These applications can be implemented using a *map* that runs using stateless functions followed by a learning stage that runs on a single multi-core server using efficient multi-core libraries [75]. The wide array of machine choices in the cloud means that this approach can handle learning problems with features up to 2TB in size [115].

As an example application we took off-the-shelf image featurization code [31] and performed cropping, scaling, and GIST image featurization [78] of the 1.28M images in the ImageNet LargeScale Visual Recognition Challenge [94]. We run the end-to-end featurization using 3000 workers on AWS Lambda. and store the features on S3. This takes 113 seconds and following that we run a monolithic *reduce* on a single `r4.16xlarge` instance. Fetching the features from S3 to this instance only takes 22s and building a linear classifier using NumPy and Intel MKL libraries takes 4.3s. Thus, we see that this model is a good fit where a high degree of parallelism is initially required to do ETL / featurization but a single node is sufficient (and most efficient [70]) for model building.

**MapReduce**: For more general purpose coordination, a commonly used programming

Table 2.2: Time taken for featurization and classification.

| phase | mean | std |
|---|---|---|
| lambda start latency | 9.7s | 29.1s |
| lambda setup time | 14.2s | 5.2s |
| featurization | 112.9s | 10.2s |
| result fetch | 22.0s | 10.0s |
| fit linear classifier | 4.3s | 0.5s |

model is the bulk-synchronous processing (BSP) model. To implement the BSP model, in addition to parallel task execution, we need to perform data shuffles across stages. The availability of high-bandwidth remote storage provides an natural mechanism to implement such shuffles. Using S3 to store shuffle data, we implemented a word count program in PyWren. On the Amazon reviews [69] dataset consisting of 83.68M product reviews split across 333 partitions, this program took 98.6s. We ran a similar program using PySpark. Using 85 `r3.xlarge` instances, each having 4 cores to match the parallelism we had with PyWren, the Spark job took 84s. The slow down is from the lack of parallel shuffle block reads in PyWren and some stragglers while writing/reading from S3. Despite that we see that PyWren is only around 17% slower than Spark and our timings do not include the 5-10 minutes it takes to start the Spark instances.

We also run the Daytona sort benchmark [102] on 1TB input, to see how PyWren handles a shuffle-intensive workload. We implemented the Terasort [77] algorithm to perform sort in two stages: a partition stage that range-partitions the input and writes out to intermediate storage, and a merge stage that, for each partition, merges and sorts all intermediate data for that partition and writes out the sorted output. Due to the resource limitation on each Lambda worker, we need at least 2500 tasks for each stage. This results in $2500^2$, or 6,250,000 intermediate files (each 160Kb) to shuffle in between. While S3 does provide abundant I/O bandwidth to Lambda for this case, it is not designed to sustain high request rate for small objects. Also as S3 is a multi-tenant service, there is an imposed limit on request throughput per S3 bucket for the benefit of overall availability. Therefore, we use S3 only for storing input and writing final output, and deploy a Redis cluster with `cache.m4.10xlarge` nodes for intermediate storage.[3] Figure 2.5 shows the end-to-end performance with varying numbers of concurrent Lambda workers and Redis shards, with breakdown of task time. We see that higher level of parallelism does greatly improve job performance (up to 500 workers) until Redis throughput becomes a bottleneck. From 500 to 1000 workers, the Redis I/O time increases by 42%. Fully leveraging this parallelism requires more Redis shards, as shown by the 44% improvement with 30 shards.

[3]Redis here can be replaced by any other key-value store, e.g., memcached, as we were only using the simple set/get API.

Interestingly, adding more resources does not necessarily increase total cost due to the reduction in latency with scale (Figure 2.6).[4] Supporting a larger sort, e.g., 100TB, does become quite challenging, as the number of intermediate files increases quadratically. We plan to investigate more efficient solutions.

**Parameter Servers**: Finally using low-latency, high throughput key-value stores like Redis, RAMCloud [93] we can also implement parameter-server [1, 63] style applications in PyWren. For example, we can implement HOGWILD! stochastic gradient descent by having each function compute the gradients based on the latest version of shared model. Since the only coordination across functions happens through the parameter server, such applications fit very well into the stateless function model. Further we can use existing support for server-side scripting [89] in key value stores to implement features like range updates and flexible consistency models [63]. However, currently this model is not easy to use as unlike S3, the ElasticCache service requires users to select a cache server type and capacity. To deploy more performant parameter servers [63] that go beyond simple key-value store would involve more complexity, e.g., setting up on a EC2 cluster or requiring a new hosted service, leaving the economic implications for further investigation.

[4] Lambda bills in 100ms increments. Redis is charged per hour and is prorated here to seconds per CloudSort benchmark rules [102].

## 2.3 Discussion

While we studied the performance provided by existing infrastructure in the previous section, there are a number of systems aspects that need to be addressed to enable high performance data processing.

**Resource Balance**: One of the primary challenges in a serverless design is in how a function's resource usage is allocated and as we mentioned in 2.2.2, the existing limits are quite low. The fact that the functions are stateless and need to transfer both input and output over the network can help cloud providers come up with some natural heuristics. For example if we consider the current constraints of AWS Lambda we see that each Lambda has around 35 MB/s bandwidth to S3 and can thus fill up its memory of 1.5GB in around 40s. Assuming it takes 40s to write output, we can see that the running time of 300s is appropriately proportioned for around 80s of I/O and 220s of compute. As memory capacity and network bandwidths grow, this rule can be used to automatically determine memory capacity given a target running time.

**Pricing** The simplicity of elastic computing comes with a premium that the users pay to the cloud providers. At the time of writing Lambda is priced at ~$0.06 per GB-hour of execution, measured in 100ms-increments. Lambda is thus only ~2× more expensive than on-demand instances. This cost premium seems worthwhile given substantially finer-grained billing, much greater elasticity, and the fact that many dedicated clusters are often running at 50% utilization. Another benefit that stems from PyWren's disaggregated architecture is that cost estimation or even cost prediction becomes much simpler. In the future we plan to explore techniques that can automatically predict the cost of a computation.

**Scalable Scheduling**: A number of cluster scheduling papers [97, 81, 82, 56] have looked at providing low latency scheduling for data parallel frameworks running on servers. However, to implement such scheduling frameworks on top of stateless functions, we need to handle the fact that information about the cluster status (i.e., which containers are free, input locations, resource heterogeneity) is only available to the infrastructure provider, while the structure of the job (i.e. how functions depend on each other) is only available to the user. In the future we plan to study what information needs to exposed by cloud providers and if scheduling techniques like offers [54] can handle this separation.

**Debugging**: Debugging can be a challenge as PyWren is composed of multiple system components. For monitoring Lambda execution we rely on service tools provided by AWS. For example, CloudWatch saves off-channel logs from Lambda workers which can be browsed through a cloud-viewer. S3 is another place to track as it contains metadata about the execution. To understand a job execution comprehensively, e.g., caclulating how much time is spent at each stage, however, would require tools to align events from both the host and Lambda.

**Distributed Storage**: With the separation of storage and compute in the PyWren programming model, a number of performance challenges translate into the need for more efficient distributed storage systems. Our benchmarks in 2.2.2 showed the limitations of

current systems, especially for supporting large shuffle-intensive workloads, and we plan to study how we can enable a flat-datacenter storage system in terms of latency and bandwidth [74]. Further, our existing benchmarks also show the limitation of not lacking API support for append in systems like S3 and we plan to develop a common API for storage backends that power serverless computation.

**Launch Overheads**: Finally one of the main drawbacks in our current implementation is that function invocation can take up to 20-30 seconds (~10% of the execution time) without any caching. This is partly due to lambda invocation rate limits imposed by AWS and partly due to the time taken to setup our custom Python runtime. We plan to study if techniques used to make VM forks cheaper [60], like caching containers or layering filesystems can be used to improve latency. We also plan to see if the scheduler can be modified to queue functions before their inputs are ready to handle launch overheads.

**Other Applications**: While we discussed data analytics applications that fit well with the serverless model, there are some applications that do not fit today. Applications that use specialized hardware like GPUs or FPGAs are not supported by AWS Lambda, but we envision that more general hardware support will be available in the future. However, for applications like particle simulations, which require a lot of coordination between long running processes, the PyWren model of using stateless functions with remote storage might not be a good fit. Finally, while we primarily focused on existing analytics applications in PyWren, the serverless model has also been used successfully in other domains like video compression [36].

# Chapter 3

# Shuffling Fast and Slow on Serverless

Serverless computing is poised to fulfill the long-held promise of transparent elasticity and millisecond-level pricing. To achieve this goal, service providers impose a fine-grained computational model where every function has a maximum duration, a fixed amount of memory and no persistent local storage. We observe that the fine-grained elasticity of serverless is key to achieve high utilization for general computations such as analytics workloads, but that resource limits make it challenging to implement such applications as they need to move large amounts of data between functions that don't overlap in time. In this Chapter, we present Locus, a serverless analytics system that judiciously combines (1) cheap but slow storage with (2) fast but expensive storage, to achieve good performance while remaining cost-efficient. Locus applies a performance model to guide users in selecting the type and the amount of storage to achieve the desired cost-performance trade-off.

## 3.1 Background

We first present a brief overview of serverless computing and compare it with the traditional VM-based instances. Next we discuss how analytics queries are implemented on serverless infrastructure and present some of the challenges in executing large scale shuffles.

### 3.1.1 Serverless Computing: What fits?

Recently, cloud providers and open source projects [51, 79] have proposed services that execute *functions* in the cloud or providing Functions-as-a-Service. As of now, these functions are subject to stringent resource limits. For example, AWS Lambda currently imposes a 5 minute limit on function duration and 3GB memory limit. Functions are also assumed to be *stateless* and are only allocated 512MB of ephemeral storage. Similar limits are applied by other providers such as Google Cloud Functions and Azure Functions. Regardless of such limitations, these offerings are popular among users for two main reasons: ease of deployment and flexible resource allocation. When deploying a cluster of virtual machines, users need to choose the instance type, number of instances, and make sure these instances are shutdown when the computation finishes. In contrast, serverless offerings have a much simpler deployment model where the functions are automatically triggered based on events, e.g., arrival of new data.

Furthermore, due to their lightweight nature, containers used for serverless deployment can often be launched within seconds and thus are easier to scale up or scale down when compared to VMs. The benefits of elasticity are especially pronounced for workloads where the number of cores required varies across time. While this naturally happens for event-driven workloads for example where say users upload a photo to a service that needs to be compressed and stored, we find that elasticity is also important for *data analytics* workloads. In particular, user-facing ad-hoc queries or exploratory analytics workloads are often unpredictable yet have more stringent responsiveness requirements, making it more difficult to provision a traditional cluster compared to recurring production workloads.

We present two common scenarios that highlight the importance of elasticitiy. First, consider a stage of tasks being run as a part of an analytics workload. As most frameworks use a BSP model [28, 116] the stage completes only when the last task completes. As the same VMs are used across stages, the cores where tasks have finished are idle while the slowest tasks or stragglers complete [3]. In comparison, with a serverless model, the cores are immediately relinquished when a task completes. This shows the importance of elasticity *within* a stage. Second, elasticity is also important *across* stages: if we consider say consider TPC-DS query 95 (details in 3.4), the query consists of 8 stages with input data per stage varying from 0.8Mb to 66Gb. With such a large variance in data size, being able to adjust the number of cores used at every stage leads to better utilization compared to traditional VM model.

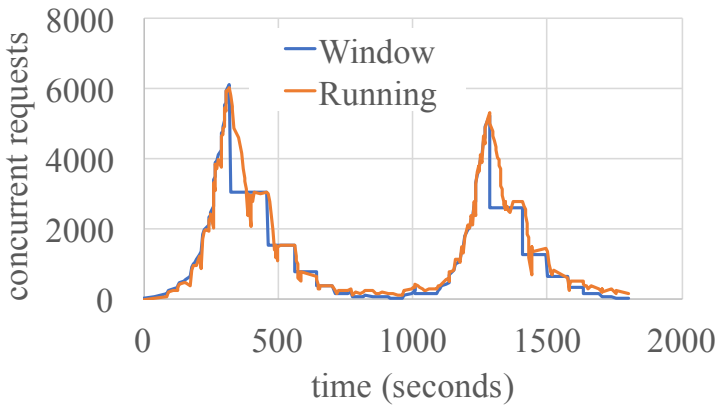



Figure 3.1: S3 rate limiting in action. We use a TCP-like additive-increase/multiplicative-decrease (AIMD) algorithm to probe the number of concurrent requests S3 can support for reading 10KB objects. We see that S3 not only enforces a rate ceiling, but also continues to fail requests after the rate is reduced for a period of time. The specific rate ceiling can change over time due to S3's automatic data-partition scaling.

### 3.1.2 Analytics on serverless: Challenges

To execute analytics queries on a serverless infrastructure we assume the following system model. A driver process, running on user's machine, "compiles" the query into a multi-stage DAG, and then submits each task to the cloud service provider. A task is executed as one function invocation by the serverless infrastructure. Tasks in consecutive stages exchange data via a variety of communication primitives, such as shuffle and broadcast [22]. Each task typically consists of three phases: read, compute, and write [80]. We next discuss why the communication between stages i.e., the shuffle stage presents the biggest challenge.

**Input, Output:** Similar to existing frameworks, each task running as a function on a serverless infrastructure reads the input from a shared storage system, such as S3. However, unlike existing frameworks, functions are not co-located with the storage, hence there is no data *locality* in this model. Fortunately, as prior work has shown, the bandwidth available between functions and the shared storage system is comparable to the disk bandwidths [4], and thus we typically do not see any significant performance degradation in this step.

**Compute:** With serverless computing platforms, each function invocation is put on a new container with a virtualized compute core. Regardless of the hardware heterogeneity, recent works have shown that the almost linear scaling of serverless compute is ideal for supporting embarrassingly parallel workloads [32, 36].

**Shuffle:** The most commonly used communication pattern to transfer data across stages is the shuffle operation. The map stage partitions data according to the number of reducers and each reducer reads the corresponding data partitions from the all the mappers. Given

$M$ mappers and $R$ reducers we will have $M*R$ intermediate data partitions. Unfortunately, the time and resource limitations imposed by the serverless infrastructures make the implementation of the shuffle operation highly challenging.

A direct approach to implementing shuffles would be to open connections between serverless workers [36] and transfer data directly between them. However, there are two limitations that prevent this approach. First cloud providers do not provide any guarantees on when functions are executed and hence the sender and receiver workers might not be executing at the same time. Second, even if the sender and receiver overlap, given the execution time limit, there might not be enough time to transfer all the necessary data.

A natural approach to transferring data between ephemeral workers is to store intermediate data in a persistent storage system. We illustrate challenges for this approach with a distributed sorting example.

### 3.1.3 Scaling Shuffle: CloudSort Example

The main challenge in executing shuffles in a serverless environment is handling the large number of intermediate files being generated. As discussed before, functions have stringent resource limitations and this effectively limits the amount of data a function can process in one task. For example to sort 100TB, we will need to create a large number of map partitions, as well as a large number of reduce partitions, such that the inputs to the tasks can be less than the memory footprint of a function. Assuming 1GB partitions, we have $10^5$ partitions on both the map side and the reduce side. For implementing a hash-based shuffle one intermediate file is created for each (mapper, reducer) pair. In this case we will have a total of $10^{10}$, or *10 billion* intermediate files! Even with traditional cluster-based deployment, shuffling 10 billion files is quite challenging, as it requires careful optimization to achieve high network utilization [77]. Unfortunately, none of the storage systems offered by existing cloud providers meets the performance requirements, while also being cost-effective. We next survey two widely available storage systems classes and discuss their characteristics.

### 3.1.4 Cloud Storage Systems Comparison

To support the diverse set of cloud applications, cloud providers offer a number of storage systems each with different characteristics in terms of latency, throughput, storage capacity and elasticity. Just as within a single machine, where we have a storage hierarchy of cache, memory and disk, each with different performance and cost points, we observe that a similar hierarchy can be applied to cloud storage systems. We next categorize two major storage system classes.

**Slow Storage**: All the popular cloud providers offer support for scalable and elastic blob storage. Examples of such systems include Amazon S3, Google Cloud Storage, Azure Blob Store. However, these storage systems are not designed to support high throughput on reading and writing small files. In fact, all major public cloud providers impose a global transaction limit on shared object stores [95, 13, 39]. This should come as no

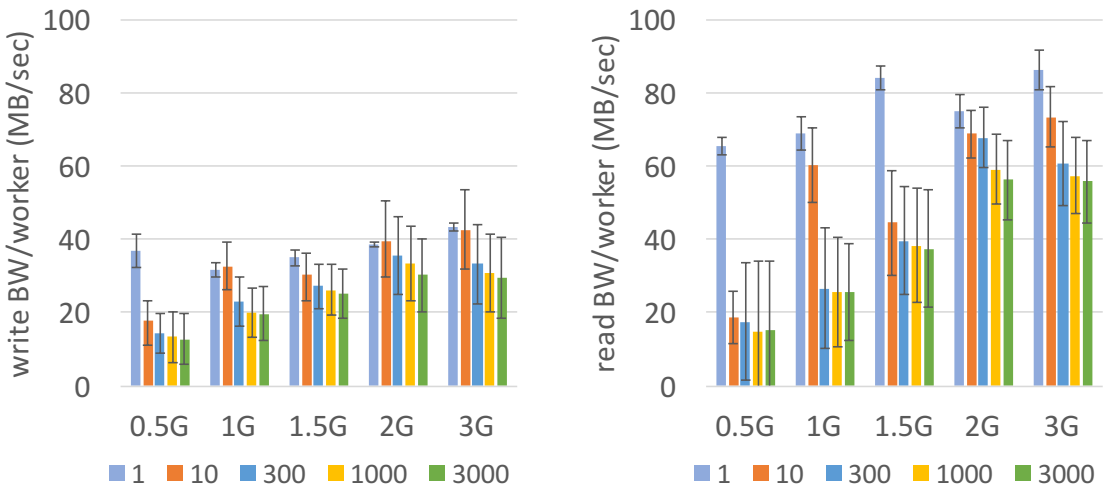
surprise, as starting with the Google File System [40], the majority of large scale storage systems have been optimized for reading and writing large chunks of data, rather than for high-throughput fine-grained operations.

We investigated the maximum throughput that one can achieve on Amazon S3 and found that though the throughput can be improved as the number of buckets increases, the cloud provider throttles requests when the aggregate throughput reaches a few thousands of requests/sec (see Figure 3.1). Assuming a throughput of 10K operations per second, this means that reading and writing all the files generated by our CloudSort example could take around 2M seconds, or 500× slower than the current record [114]. Not only is the performance very low, but the cost is prohibitive as well. While the cost per write request is as low as $0.005 per 1,000 requests for all three aforementioned cloud providers, shuffling $10^{10}$ files would cost $5,000 alone for write requests. Thus, supporting large shuffles requires a more efficient and economic solution for storing intermediate data.

**Fast Storage**: One approach to overcome the performance limitations of the slow storage systems is to use much faster storage, if available. Examples of faster storage are in-memory storage systems backed by Memcached or Redis. Such storage systems support much higher request rates (more than 100,000 requests/sec per shard), and efficiently handle objects as small as a few tens of bytes. On the flip side, these systems are typically much more expensive than large-scale blob storage systems. For example to store 1GB of data for an hour, it costs 0.00319 cents in AWS S3 while it costs 2.344 cents if we use a managed Redis service such as AWS ElastiCache, which makes it 733× more expensive![1]

Given the cost-performance trade-off between slow (e.g., S3) and fast (e.g., ElastiCache) storage, in the following sections we show that by judiciously combining these two types of storage systems, we can achieve a cost-performance sweet spot in a serverless deployment that is comparable, and sometimes superior to cluster-based deployments.

[1]We note that ElastiCache is not "serverless", and there is no serverless cache service yet as of writing this thesis and users need to provision cache instances. However, we envision that similar to existing storage and compute, fast storage as a resource (possibly backed by memory) will also become elastic in the future. There are already several proposals to provide disaggregated memory across datacenters [38] to support this.




(a) write (b) read

Figure 3.2: S3 bandwidth per worker with varying concurrency (1 to 3000) and Lambda worker size (0.5G to 3G).

Table 3.1: Measured throughput (requests/sec) limit for a single S3 bucket and a single Redis shard.

| object size | 10KB | 100KB | 1M | 10M | 100M |
|---|---|---|---|---|---|
| S3 | 5986 | 4400 | 3210 | 1729 | 1105 |
| Redis | 116181 | 11923 | 1201 | 120 | 12 |

## 3.2 Design

In this section we outline a performance model that can be used to guide the design of an efficient and cost-effective shuffle operations. We start with outlining our system model, and then discuss how different variables like worker memory size, degree of parallelism, and the type of storage system affect the performance characteristics of the shuffle operation.

### 3.2.1 System Model

We first develop a high level system model that can be used to compare different approaches to shuffle and abstract away details specific to cloud providers. We denote the function-as-a-service module as *compute cores* or *workers* for tasks. Each function invocation, or a worker, is denoted to run with a single core and $w$ bytes of memory (or the worker memory size). The degree of the parallelism represents the number of function

Table 3.2: Cloud storage cost from major providers (Feb 2019).

| | Service | $/Mo/GB | $/million writes |
|---|---|---|---|
| Slow | AWS S3 | 0.023 | 5 |
| | GCS | 0.026 | 5 |
| | Azure Blob | 0.023 | 6.25 |
| Fast | ElastiCache | 7.9 | - |
| | Memorystore | 16.5 | - |
| | Azure Cache | 11.6 | - |

invocations or workers that execute in parallel, which we denote by $p$. The total amount of data being shuffled is $S$ bytes. Thus, the number of workers required in the map and reduce phase is at least $\frac{S}{w}$ leading to a total of $(\frac{S}{w})^2$ requests for a full shuffle.

We next denote the bandwidth available to access a storage service by an individual worker as $b$ bytes/sec. We assume that the bandwidth provided by the elastic storage services scale as we add more workers (we discuss how to handle cases where this is not true below). Finally, we assume each storage service limits the aggregate number of requests/sec: we denote by $q_s$ and $q_f$ for the slow and the fast storage systems, respectively.

To measure the cost of each approach we denote the cost of a worker function as $c_l$ $/sec/byte, the cost of fast storage as $c_f$ $/sec/byte. The cost of slow storage has two parts, one for storage as $c_s$ $/sec/byte, and one for access, denoted as $c_a$ $/op. We assume that both the inputs and the outputs of the shuffle are stored on the slow storage. In most cases in practice, $c_s$ is negligible during execution of a job. We find the above cost characteristics apply to all major cloud platforms (AWS, Google Cloud and Azure), as shown in Table 3.2.

Among the above, we assume the shuffle size ($S$) is given as an input to the model, while the worker memory size ($w$), the degree of parallelism ($p$), and the amount of fast storage ($r$) are the model knobs we vary. To determine the characteristics of the storage systems (e.g., $b$, $q_s$, $q_f$), we use offline benchmarking. We first discuss how these storage performance characteristics vary as a function of our variables.

### 3.2.2 Storage Characteristics

The main storage characteristics that affect performance are unsurprisingly the read and write throughput (in terms of requests/sec, or often referred as IOPS) and bandwidth (in terms of bytes/sec). However, we find that these values are not stable as we change the degree of parallelism and worker memory size. In Figure 3.2 we measure how a function's bandwidth ($b$) to a large-scale store (i.e., Amazon S3, the slow storage service in our case) varies as we change the degree of parallelism ($p$) and the worker memory size ($w$). From the figure we can see that as we increase the parallelism both read and write bandwidths could vary by 2-3×. Further we see that as we increase the worker memory

Table 3.3: Comparison of time taken by different shuffle methods. $S$ refers to the shuffle data size, $w$ to the worker memory size, $p$ the number of workers, $q_s$ the throughput to slow storage, $q_f$ throughput to fast storage $b$ network bandwidth from each worker.

| storage type | shuffle time |
|---|---|
| slow | $2 \times max(\frac{S^2}{w^2 \times q_s}, \frac{S}{b \times p})$ |
| fast | $2 \times max(\frac{S^2}{w^2 \times q_f}, \frac{S}{b_{eff}})$, where<br>$b_{eff} = min(b_f, b \times p)$ |
| hybrid | $\frac{S}{r} T_{rnd} + T_{mrg}$, where<br>$T_{rnd} = 2 \times max(T_{fb}, T_{sb}, T_{sq})$<br>$T_{mrg} = 2 \times max((\frac{Sw}{r})^2 T_{sq}, \frac{S}{r} T_{sb})$<br>$T_{fb} = \frac{r}{b_{eff}}, T_{sb} = \frac{r}{b \times p}$<br>$T_{sq} = \frac{r^2}{w^2 \times q_s}$ |

size the bandwidth available increases but that the increase is sub-linear. For example with 60 workers each having 0.5G of memory, the write bandwidth is around 18 MB/s per worker or 1080 MB/s in aggregate. If we instead use 10 workers each having 3GB of memory, the write bandwidth is only around 40 MB/s per worker leading to 400 MB/s in aggregate.

Using a large number of small workers is not always ideal as it could lead to an increase in the number of small I/O requests. Table 3.1 shows the throughput we get as we vary the object size. As expected, we see that using smaller object sizes means that we get a lower aggregate bandwidth (multiplying object size by transaction throughput). Thus, jointly managing worker memory size and parallelism poses a challenging trade-off.

For fast storage systems we typically find that throughput is not a bottleneck for object sizes > 10 KB and that we saturate the storage bandwidth. Hence, as shown in Table 3.1 the operation throughput decreases linearly as the object size increases. While we can estimate the bandwidth available for fast storage systems using an approach similar to the one used for slow storage systems, the current deployment method where we are allocating servers for running Memcached / Redis allows us to ensure they are not a bottleneck.

### 3.2.3 Shuffle Cost Models

We next outline performance models for three shuffle scenarios: using (1) slow storage only, (2) fast storage only, and (3) a combination of fast and slow storage.

**Slow storage based shuffle**. The first model we develop is using slow storage only to perform the shuffle operation. As we discussed in the previous section there are two limits that the slow storage systems impose: an operation throughput limit ($q_s$) and

a bandwidth limit ($b$). Given that we need to perform $(\frac{S}{w})^2$ requests with an overall operation throughput of $q_s$, we can derive $T_q$, the time it takes to complete these requests is $T_q = \frac{S^2}{w^2 \times q_s}$, assuming $q_s$ is the bottleneck. Similarly, given the per-worker bandwidth limit to storage, $b$, the time to complete all requests assuming $b$ is bottleneck is $T_b = \frac{S}{b \times p}$. Considering both potential bottlenecks, the time it takes to write/read all the data to/from intermediate storage is thus $\max(T_q, T_b)$. Note that this time already includes reading data from input storage or writing data to output storage, since they can be pipelined with reading/writing to intermediate storage. Finally, the shuffle needs to first write data to storage and then read it back. Hence the total shuffle time is $T_{shuf} = 2 \times \max(T_q, T_b)$.

Table 3.4 shows our estimated running time and cost as we vary the worker memory and data size.

**Fast storage based-shuffle**. Here we develop a simple performance model for fast storage that incorporates the throughput and bandwidth limits. In practice we need to make one modification to factor in today's deployment model for fast storage systems. Since services like ElastiCache are deployed by choosing a fixed number of instances, each having some fixed amount of memory, the aggregate bandwidth of the fast storage system could be a significant bottleneck, if we are not careful. For example, if we had just one ElastiCache instance with 10Gbps NIC and 50G of memory, the aggregate bandwidth is trivially limited to 10Gbps. In order to model this aspect, we extend our formulation to include $b_f$, which is the server-side bandwidth limit for fast storage. We calculate the effective bandwidth as $b_{eff} = \min(b \times p, b_f)$.

Using the above effective bandwidth we can derive the time taken due to throughput and bandwidth limits as $T_q = \frac{S^2}{w^2 \times q_f}$ and $T_b = \frac{S}{b_{eff}}$, respectively. Similar to the previous scenario, the total shuffle time is then $T_{shuf} = 2 \times max(T_q, T_b)$.

One interesting scenario in this case is that as long as the fast storage bandwidth is a bottleneck (i.e. $b_f < b \times p$), using more fast memory improves not only the performance, but also reduces the cost! Assume the amount of fast storage is $r$. This translates to a cost of $p * c_l * T_{shuf} + r * c_f * T_{shuf}$, with slow storage request cost excluded. Now, assume we double the memory capacity to $2 \times r$, which will also result in doubling the bandwidth, i.e., $2 \times b_f$. Assuming that operation throughput is not the bottleneck, the shuffle operations takes now $\frac{S}{2b_f} = \frac{T_{shuf}}{2}$, while the cost becomes $p * c_l * \frac{T_{shuf}}{2} + 2 * r * c_f * \frac{T_{shuf}}{2}$. This does not include reduction in request cost for slow storage. Thus, while the cost for fast storage (second term) remains constant, the cost for compute cores drops by a factor of 2. In other words, the overall running time has improved by a factor of 2 while the cost has decreased.

However, as the amount of shuffle data grows, the cost of storing all the intermediate data in fast storage becomes prohibitive. We next look at the design of a hybrid shuffle method that can scale to much larger data sizes.

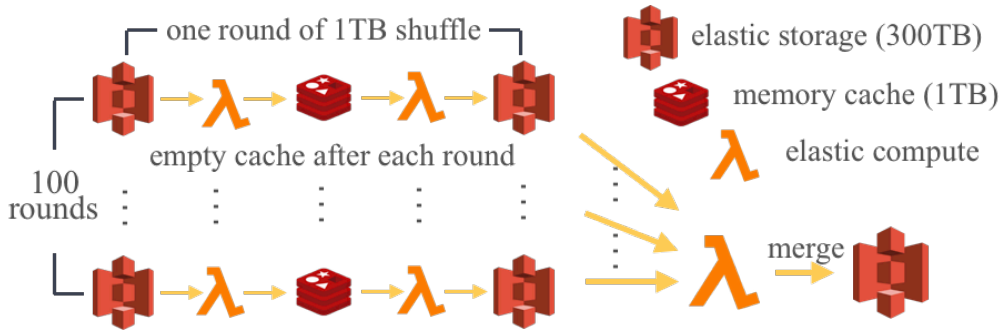



Figure 3.3: Illustration for hybrid shuffle.

Table 3.4: Projected sort time and cost with varying worker memory size. Smaller worker memory results in higher parallelism, but also a larger numbers files to shuffle.

| worker mem(GB) | 0.25 | 0.5 | 1 | 1.25 | 1.5 |
|---|---|---|---|---|---|
| 20GB time(s) | 36 | 45 | 50 | 63 | 72 |
| 20GB cost($) | 0.02 | 0.03 | 0.03 | 0.04 | 0.05 |
| 200GB time(s) | 305 | 92 | 50 | 63 | 75 |
| 200GB cost($) | 0.24 | 0.30 | 0.33 | 0.42 | 0.51 |
| 1TB time(s) | 6368 | 1859 | 558 | 382 | 281 |
| 1TB cost($) | 1.22 | 1.58 | 1.70 | 2.12 | 2.54 |

### 3.2.4 Hybrid Shuffle

We propose a hybrid shuffle method that combines the inexpensive slow storage with the high throughput of fast storage to reach a better cost-performance trade-off. We find that even with a small fast storage, e.g., less than $\frac{1}{20}$th of total shuffle data, our hybrid shuffle can outperform slow storage based shuffle by orders of magnitude.

To do that, we introduce a multi-round shuffle that uses fast storage for intermediate data within a round, and uses slow storage to merge intermediate data across rounds. In each round we range-partition the data into a number of buckets in fast storage and then combine the partitioned ranges using the slow storage. We reuse the same range partitioner across rounds. In this way, we can use a merge stage at the end to combine results across all rounds, as illustrated in Figure 3.3. For example, a 100 TB sort can be broken down to 100 rounds of 1TB sort, or 10 rounds of 10TB sort.

Correspondingly the cost model for the hybrid shuffle can be broken down into two parts: the cost per round and the cost for the merge. The size of each round is fixed at $r$, the amount of space available on fast storage. In each round we perform two stages of computation, partition and combine. In the partition stage, we read input data from the slow storage and write to the fast storage, while in the combine stage we read from the fast

storage and write to the slow storage. The time taken by one stage is then the maximum between the corresponding durations of the stage when the bottleneck is driven either by (1) the fast storage bandwidth $T_{fb} = \frac{r}{b_{eff}}$, (2) the slow storage bandwidth $T_{sb} = r/(b * p)$, or (3) the slow storage operation throughput $T_{sq} = \frac{r^2}{w^2 \times q_s}$[2]. Thus, the time per-round is $T_{rnd} = 2 * max(T_{fb}, T_{sb}, T_{sq})$.

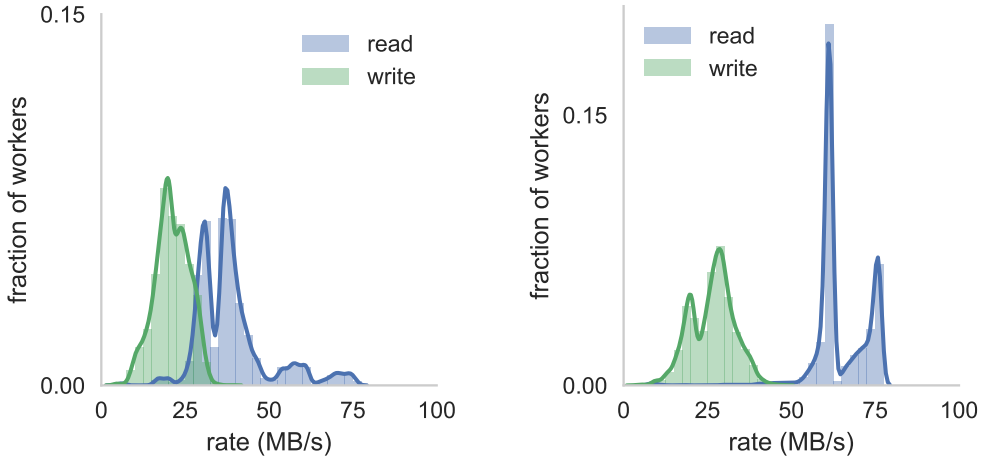
The overall shuffle consists of $\frac{S}{r}$ such rounds and a final merge phase where we read data from the slow storage, merge it, and write it back to the slow storage. The time of the merge phase can be similarly broken down into throughput limit $T_{mq} = (\frac{Sw}{r})^2 * T_{sq}$ and bandwidth limit $T_{mb} = \frac{S}{r} * T_{sb}$, where $T_{sb}$ and $T_{sq}$ follows from the definitions from previous paragraph. Thus, $T_{mrg} = 2 * max(T_{mq}, T_{mb})$, and the total shuffle time is $\frac{S}{r} * T_{rnd} + T_{mrg}$.

**How to pick the right fast storage size?** Selecting the appropriate fast storage/memory size is crucial to obtaining good performance with the hybrid shuffle. Our performance model aims to determine the optimal memory size by using two limits to guide the search. First, provisioning fast storage does not help when slow storage bandwidth becomes bottleneck, which provides an upper bound on fast storage size. Second, since the final stage needs to read outputs from all prior rounds to perform the merge, the operation throughput of the slow storage provides an upper bound on the number of rounds, thus a lower bound of the fast storage size.

**Pipelining across stages** An additional optimization we perform to speed up round execution and reduce cost is to pipeline across partition stage and combine stage. As shown in Figure 3.3, for each round, we launch partition tasks to read input data, partition them and write out intermediate files to the fast storage. Next, we launch combine tasks that read files from the fast storage. After each round, the fast storage can be cleared to be used for next round.

With pipelining, we can have partition tasks and combine tasks running in parallel. While the partition tasks are writing to fast storage via `append()`, the merge tasks read out files periodically and perform atomic delete-after-read operations to free space. Most modern key-value stores, e.g., Redis, support operations such as append and atomic delete-after-read. Pipelining gives two benefits: (1) it overlaps the execution of the two phases thus speeding up the in-round sort, and (2) it allows a larger round size without needing to store the entire round in memory. Pipelining does have a drawback. Since we now remove synchronization boundary between rounds, and use `append()` instead of setting a new key for each intermediate data, we cannot apply speculative execution to mitigate stragglers, nor can we obtain task-level fault tolerance. Therefore, pipelining is more suitable for smaller shuffles.

[2]We ignore the fast storage throughput, as we rarely find it to be bottleneck. We could easily include it in our model, if needed.




(a) 500MB workers (b) 3GB workers

Figure 3.4: Lambda to S3 bandwidth distribution exhibits high variance. A major source of stragglers.

### 3.2.5 Modeling Stragglers

The prior sections provided several basic models to estimate the time taken by a shuffle operation in a serverless environment. However, these basic models assume all tasks have uniform performance, thus failing to account for the presence of stragglers.

The main source of stragglers for the shuffle tasks we consider in this work are network stragglers, that are caused by slow I/O to object store. Network stragglers are inherent given the aggressive storage sharing implied by the serverless architecture. While some containers (workers) might get better bandwidth than running reserved instances, some containers get between 4-8× lower bandwidth, as shown in Figure 3.4. To model the straggler mitigation scheme described above we initialize our model with the network bandwidth CDFs as shown in Figure 3.4. To determine running time of each stage we then use an execution simulator [80] and sample network bandwidths for each container from the CDFs. Furthermore, our modeling is done for each worker memory size, since bandwidth CDFs vary across worker sizes.

There are many previous works on straggler mitigation [118, 7, 92, 5]. We use a simple online method where we always launch speculative copies after $x$% of tasks finish in the last wave. Having short-lived tasks in the serverless model is more advantageous here. The natural elasticity of serverless infrastructure makes it possible to be aggressive in launching speculative copies.

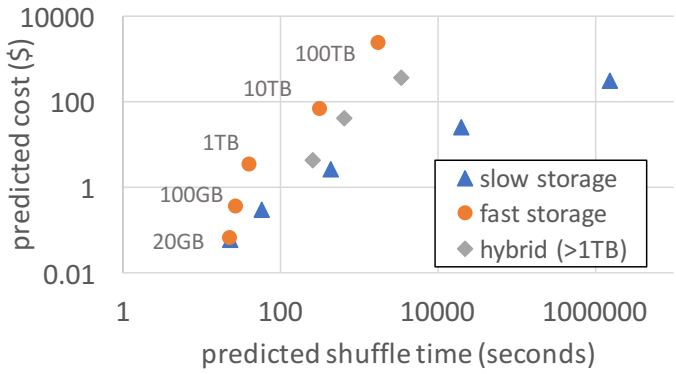



Figure 3.5: Predicted time and cost for different sort implementations and sizes.

### 3.2.6 Performance Model Case Study

We next apply our performance model described above to the CloudSort benchmark and study the cost-performance trade-off for the three approaches described above. Our predictions for data sizes ranging from 20GB to 100TB are shown in Figure 3.5 (we use experimental results of a real prototype to validate these predictions in Section 3.4). When the data shuffle size is small (e.g., 20GB or smaller), both the slow and fast storage only solutions take roughly the same time, with the slow storage being slightly cheaper. As the data size increases to around 100GB, using fast storage is around 2× faster for the same cost. This speed up from fast storage is more pronounced as data size grows. For very large shuffles ($\geq$ 10 TB), hybrid shuffle can provide significant cost savings. For example, at 100TB, the hybrid shuffle is around $6x$ cheaper than the fast storage only shuffle, but only $2x$ slower.

Note that since the hybrid shuffle performs a merge phase in addition to writing all the data to the fast storage, it is always slower than the fast storage only shuffle. In summary, this example shows how our performance model can be used to understand the cost-performance trade-off from using different shuffle implementations. We implement this performance modeling framework in Locus to perform automatic shuffle optimization. We next describe the implementation of Locus and discuss some extensions to our model.

## 3.3 Implementation

We implement Locus by extending PyWren [32], a Python-based data analytics engine developed for serverless environments. PyWren allows users to implement custom functions that perform data shuffles with other cloud services, but it lacks an actual shuffle operator. We augment PyWren with support for shuffle operations and implement the performance modeling framework described before to automatically configure the shuffle variables. For our implementation we use AWS Lambda as our compute engine and use S3 as the slow, elastic storage system. For fast storage we provision Redis nodes on Amazon ElastiCache.

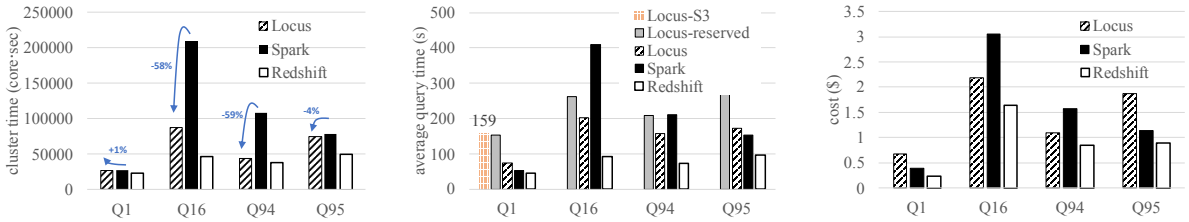
To execute SQL queries on Locus, we devise physical query plan from Apache Spark and then use Pandas to implement structured data operations. One downside with Pandas is that we cannot do "fine-grained pipelining" between data operations inside a task. Whereas in Apache Spark or Redshift, a task can process records as they are read in or written out. Note this fine-grained pipelining is different from pipelining across stages, which we discuss in Section 3.2.4.

### 3.3.1 Model extensions

We next discuss a number of extensions to augment the performance model described in the previous section

**Non-uniform data access**: The shuffle scenario we considered in the previous section was the most general all-to-all shuffle scenario where every mapper contributes data to every reducer. However, a number of big data workloads have more skewed data access patterns. For example, machine learning workloads typically perform `AllReduce` or `broadcast` operations that are implemented using a tree-based communication topology. When a binary tree is used to do AllReduce, each mapper only produces data for one reducer and correspondingly each reducer only reads two partitions. Similarly while executing a broadcast join, the smaller table will be accessed by every reducer while the larger table is hash partitioned. Thus, in these scenarios storing the more frequently accessed partition on fast storage will improve performance. To handle these scenarios we introduce an access counter for each shuffle partition and correspondingly update the performance model. We only support this currently for cases like AllReduce and broadcast join where the access pattern is known beforehand.

**Storage benchmark updates**: Finally one of the key factors that make our performance models accurate is the storage benchmarks that measure throughput (operations per sec) and network bandwidth (bytes per second) of each storage system. We envision that we will execute these benchmarks the first time a user installs Locus and that the benchmark values are reused across a number of queries. However, since the benchmarks are capturing the behavior of cloud storage systems, the performance characteristics could change over time. Such limits change will require Locus to rerun the profiling. We plan to investigate techniques where we can profile query execution to infer whether our benchmarks are still accurate over extended periods of time.




(a) Cluster time (b) Query latency (c) Cost

Figure 3.6: TPC-DS results for Locus, Apache Spark and Redshift under different configurations. Locus-S3 runs the benchmark with only S3 and doesn't complete for many queries; Locus-reserved runs Locus on a cluster of VMs.

## 3.4 Evaluation

We evaluate Locus with a number of analytics workloads, and compare Locus with Apache Spark running on a cluster of VMs and AWS Redshift/Redshift Spectrum[3]. Our evaluation shows that:

- Locus's serverless model can reduce cluster time by up to 59%, and at the same time being close to or beating Spark's query completion time by up to 2×. Even with a small amount of fast storage, Locus can greatly improve performance. For example, with just 5% memory, we match Spark in running time on CloudSort benchmark and are within 13% of the cost of the winning entry in 2016.
- When comparing with actual experiment results, our model in Section 3.2 is able to predict shuffle performance and cost accurately, with an average error of 15.9% for performance and 14.8% for cost. This allows Locus to choose the best cost-effective shuffle implementation and configuration.
- When running data intensive queries on the same number of cores, Locus is within 1.61 × slower compared to Spark, and within 2 × slower compared to Redshift, regardless of the baselines' more expensive unit-time pricing. Compared to shuffling only through slow storage, Locus can be up to 4×-500× faster.

The section is organized as follows, we first show utilization and end-to-end performance with Locus on TPC-DS [86] queries ( 3.4.1) and Daytona CloudSort benchmark ( 3.4.2). We then discuss how fast storage shifts resource balance to affect the cost-performance trade-off in Section 3.4.3. Using the sort benchmark, we also check whether our shuffle formulation in Section 3.2 can accurately predict cost and performance( 3.4.4). Finally we evaluate Locus's performance on joins with Big Data Benchmark [15]( 3.4.5).

[3]When reading data of S3, AWS Redshift automatically uses a shared, serverless pool of resource called the Spectrum layer for S3 I/O, ETL and partial aggregation.

**Setup**: We run our experiments on AWS Lambda and use Amazon S3 for slow storage. For fast storage, we use a cluster of `r4.2xlarge` instances (61GB memory, up to 10Gbps network) and run Redis. For our comparisons against Spark, we use the latest version of Apache Spark (2.3.1). For comparison against Redshift, we use the latest version as of 2018 September and `ds2.8xlarge` instances. To calculate cost for VM-based experiments we pro-rate the hourly cost to a second granularity.[4] For Redshift, the cost is two parts using AWS pricing model, calculated by the uptime cost of cluster VMs, plus $5 per TB data scanned.

### 3.4.1 TPC-DS Queries

The TPC-DS benchmark has a set of standard decision support queries based on those used by retail product suppliers. The queries vary in terms of compute and network I/O loads. We evaluate Locus on TPC-DS with scale factor of 1000, which has a total input size of 1TB data for various tables. Among all queries, we pick four of them that represent different performance characteristics and have a varying input data size from 33GB to 312GB. Our baselines are Spark SQL deployed on a EC2 cluster with `c3.8xlarge` instances and Redshift with `ds2.8xlarge` instances, both with 512 cores. For Locus, we obtain workers dynamically across different stages of a query, but make sure that we never use more core·secs of Spark execution.

Figure 3.6(b) shows the query completion time for running TPC-DS queries on Apache Spark, Redshift and Locus under different configurations and Figure 3.6(a) shows the the total core·secs spent on running those queries. We see that Locus can save cluster time up to 59%, while being close to Spark's query completion time to also beating it by 2×. Locus loses to Spark on Q1 by 20s. As a result, even for now AWS Lambda's unit time cost per core is 1.92× more expensive than the EC2 `c3.8xlarge` instances, Locus enjoys a lower cost for Q1 and Q4 as we only allocate as many Lambdas as needed. Compared to Redshift, Locus is 1.56× to 1.99× slower. There are several causes that might contribute to the cost-performance gap: 1) Redshift has a more efficient execution workflow than that of Locus, which is implemented in Python and has no fine-grained pipelining; 2) `ds2.8xlarge` are special instances that have 25Gbps aggregate network bandwidths; 3) When processing S3 data, AWS Redshift pools extra resource, referred as the serverless Spectrum layer, to process S3 I/O, ETL and partial aggregation. To validate these hypotheses, we perform two what-if analyses. We first take Locus's TPC-DS execution trace and replay them to numerically simulate an pipelined execution by overlapping I/O and compute within a task. We find that with pipelining, query latencies can be reduced by 23% to 37%, being much closer to the Redshift numbers. Similarly, using our cost-performance model, we also find that if Locus's Redis nodes have 25Gbps links, the cost can be further reduced by 19%, due to a smaller number of nodes needed. Performance will not improve due to 25Gbps links, as network bottleneck on Lambda-side remains. Understanding remaining

[4]This is presenting a lower cost than the minute-granularity used for billing by cloud providers like Amazon, Google.

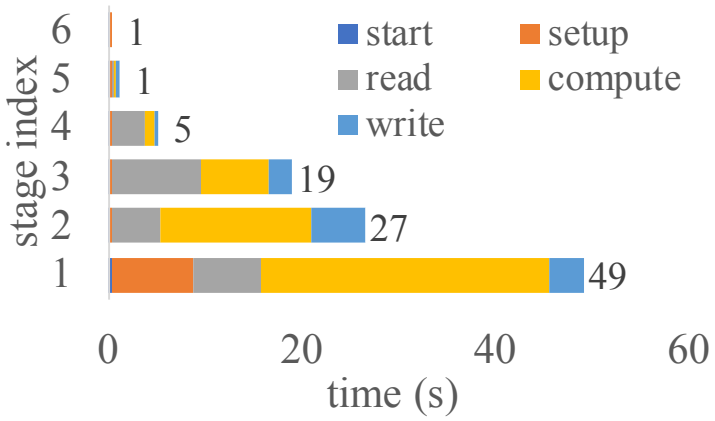



Figure 3.7: Time breakdown for Q94. Each stage has a different profile and, compute and network time dominate.

performance gap would require further breakdown, i.e., porting Locus to a lower-level programming language.

Even with the performance gap, an user may still prefer Locus over a data warehousing service like Redshift since the latter requires on-demand provisioning of a cluster. Currently with Amazon Redshift, provisioning a cluster takes minutes to finish, which is longer than these TPC-DS query latencies. Picking an optimal cluster size for a query is also difficult without knowledge of underlying data.

We also see in Figure 3.6(b) that Locus provides better performance than running on a cluster of 512-core VMs (Locus-reserved). This demonstrates the power of elasticity in executing analytics queries. Finally, using the fast storage based shuffle in Locus also results in successful execution of 3 queries that could not be executed with slow storage based shuffle, as the case for Locus-S3 or PyWren.

To understand where time is spent, we breakdown execution time into different stages and resources for Q94, as shown in Figure 3.7. We see that performing compute and network I/O takes up most of the query time. One way to improve overall performance given this breakdown is to do "fine-grained pipelining" of compute and network inside a task. Though nothing fundamental, it is unfortunately difficult to implement with the constraints of Pandas API at the time of writing. Compute time can also be improved if Locus is prototyped using a lower-level language such as C++.

Finally, for shuffle intensive stages such as stage 3 of Q94, we see that linearly scaling up fast storage does linearly improve shuffle performance (Figure 3.8).

### 3.4.2 CloudSort Benchmark

We run the Daytona CloudSort benchmark to compare Locus against both Spark and Redshift on reserved VMs.

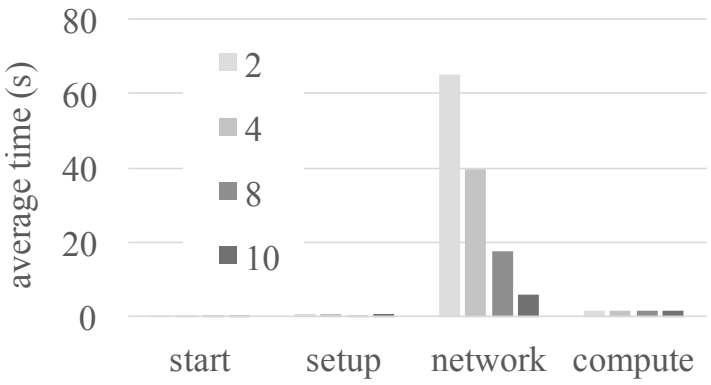



Figure 3.8: Runtime for stage 3 of Q94 when varying the number of Redis nodes (2, 4, 8, 10).

The winner entry of CloudSort benchmark which ranks the cost for sorting 100TB data on public cloud is currently held by Apache Spark [114]. The record for sorting 100TB was achieved in 2983.33s using a cluster of 395 VMs, each with 4 vCPU cores and 8GB memory. The cost of running this was reported as $144.22. To obtain Spark numbers for 1TB and 10TB sort sizes, we varied the number of `i2.8xlarge` instances until the sort times matched those obtained by Locus. This allows a fair comparison on the cost. As discussed in Section 3.2, Locus automatically picks the best shuffle implementation for each input size.

Table 3.5 shows the result cost and performance comparing Locus against Spark. We see that regardless of the fact Locus's sort runs on memory-constrained compute infrastructure and communicates through remote storage, we are within 13% of the cost for 100TB record, and achieve the same performance. Locus is even cheaper for 10TB (by 15%) but is 73% more expensive for 1TB. This is due to using fast storage based-shuffle which yields a more costly trade-off point. We discuss more trade-offs in Section 3.4.3.

Table 3.6 shows the result of sorting 1TB of random string input. Since Redshift does not support querying against random binary data, we instead generate random string records as the sort input as an approximation to the Daytona CloudSort benchmark. For fair comparison, we also run other systems with the same string dataset. We see that Locus is an order of magnitude faster than Spark and Redshift and is comparable to Spark when input is stored on local disk.

We also run the same Locus code on EC2 VMs, in order to see the cost vs. performance difference of only changing hardware infrastructure while using the same programming language (Python in Locus). Figure 3.9 shows the results for running 100GB sort. We run Locus on AWS Lambda with various worker memory sizes. Similar to previous section, we then run Locus on a cluster and vary the number of `c1.xlarge` instances to match the performance and compare the cost. We see that both cost and performance improves for Locus-serverless when we pick a smaller memory size. The performance improvement

Table 3.5: CloudSort results vs. Apache Spark.

| Sort size | 1TB | 10TB | 100TB |
|---|---|---|---|
| Spark nodes | 21 | 60 | 395[77] |
| Spark time (s) | 40 | 394 | 2983 |
| Locus time (s) | 39 | 379 | 2945 |
| Spark cost ($) | 1.5 | 34 | 144 |
| Locus cost ($) | 2.6 | 29 | 163 |

Table 3.6: 1TB string sort w/ various configurations.

| | time | cost($) |
|---|---|---|
| Redshift-S3 | 6m8s | 20.2 |
| Spark RDD-S3 | 4m27s | 15.7 |
| Spark-HDFS ($) | 35s | 2.1 |
| Locus ($) | 39s | 2.6 |

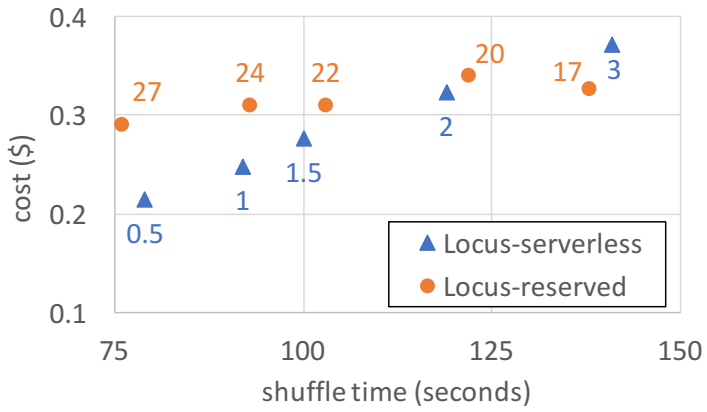



Figure 3.9: Running 100GB sort with Locus on a serverless infrastructure vs. running the same code on reserved VMs. Labels for serverless series represents the configured memory size of each Lambda worker. Labels for reserved series represents the number of c1.xlarge instances deployed.

is due to increase in parallelism that results in more aggregate network bandwidth. The cost reduction comes from both shorter run-time and lower cost for small memory sizes. For Locus-reserved, performance improves with more instances while the cost remains relatively constant, as the reduction in run-time compensates for the increased allocation.

We see that even though AWS Lambda is considered to be more expensive in terms

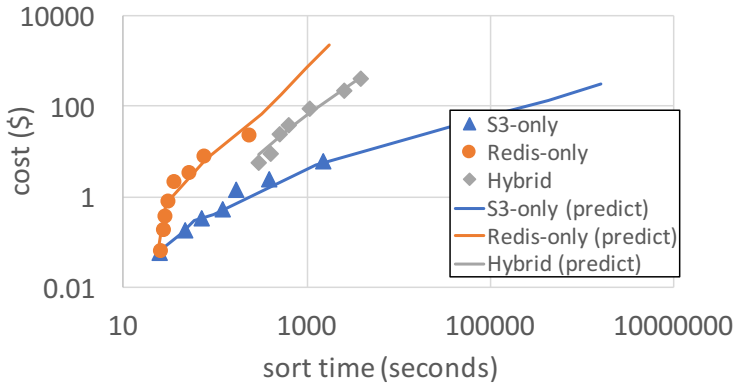



Figure 3.10: Comparing the cost and performance predicted by Locus against actual measurements. The lines indicate predicted values and the dots indicate measurements.

of $ per CPU cycle, it can be cheaper in terms of $ per Gbps compared to reserved instances. Thus, serverless environments can reach a better cost performance point for network-intensive workloads.

### 3.4.3 How Much Fast Storage is Needed?

One key insight in formulating the shuffle performance in Locus is that adding more resources does not necessarily increase total cost, e.g., increasing parallelism can result in a better configuration. Another key insight is that using fast storage or memory, sometimes even a small amount, can significantly shift resource balance and improve performance.

We highlight the first effect with an example of increasing parallelism and hence over allocating worker memory compared to the data size being processed. Consider the case where we do a slow storage-only sort for 10GB. Here, we can further increase parallelism by using smaller data partitions than the worker memory size. We find that by say using a parallelism of 40 with 2.5G worker memory size can result in 3.21× performance improvement and lower cost over using parallelism of 10 with 2.5G worker memory (Figure 3.11).

However, such performance increase does require that we add resources in a balanced manner as one could also end up incurring more cost while not improving performance. For example, with a 100GB sort (Figure 3.12), increasing parallelism from 200 to 400 with 2.5G worker memory size (Figure 3.12) makes performance 2.5× worse, as now the bottleneck shifts to object store throughput and each worker will run slower due to a even smaller share. Compared to the 10GB sort, this also shows that the same action that helps in one configuration can be harmful in another configuration.

Another way of balancing resources here is to increase parallelism while adding fast storage. We see this in Figure 3.12, where increasing parallelism to 400 becomes beneficial with fast storage as the storage system can now absorb the increased number of requests. These results provide an example of the kinds of decisions automated by the performance

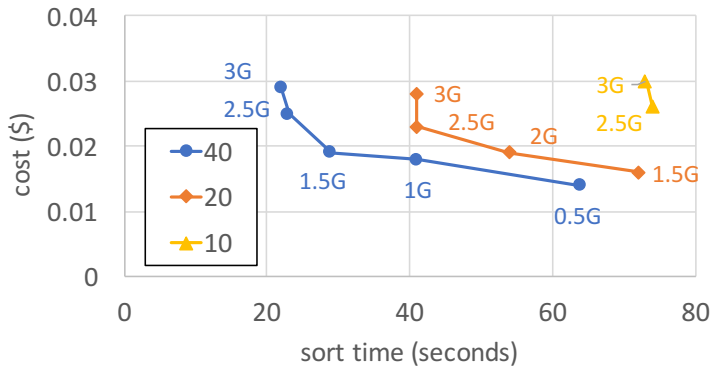



Figure 3.11: 10GB slow storage-only sort, with varying parallelism (lines) and worker memory size (dots).

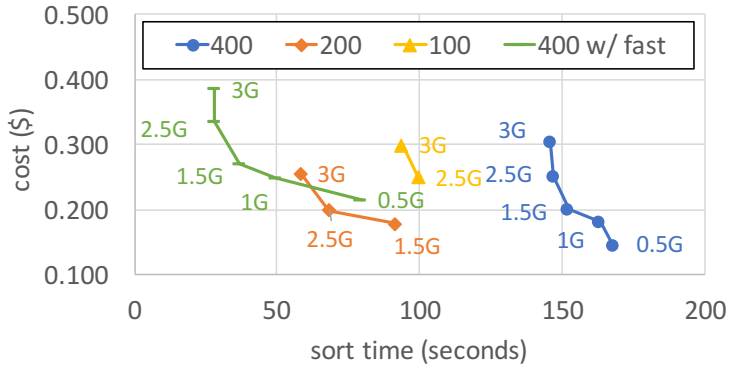



Figure 3.12: 100GB slow storage-only sort with varying parallelism (different lines) and worker memory size (dots on same line). We include one configuration with fast-storage sort.

modeling framework in Locus.

The second insight is particularly highlighted for running 100TB hybrid sort. For 100TB sort, we vary the fast storage used from 2% to 5%, and choose parallelism for each setting based on the hybrid shuffle algorithm. As shown in Table 3.7, we see that even with 2% of memory, the 100TB sort becomes attainable in 2 hours. Increasing memory from 2% to 5%, there is an almost linear reduction in terms of end-to-end sort time when we use larger cache size.This matches the projection in our design discussion. Further broken down in Figure 3.13, we see that the increase of cost per time unit is compensated by reduction in end-to-end run time.

Table 3.7: 100TB Sort with different cache size.

| cache | 5% | 3.3% | 2.5% | 2% |
|---|---|---|---|---|
| time (s) | 2945 | 4132 | 5684 | 6850 |
| total cost ($) | 163 | 171 | 186 | 179 |

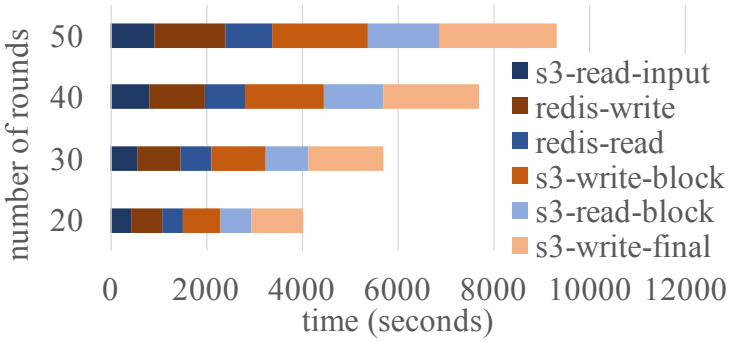



Figure 3.13: Runtime breakdown for 100TB sort.

### 3.4.4 Model Accuracy

To automatically choose a cost-effective shuffle implementation, Locus relies on a predictive performance model that can output accurate run-time and cost for any sort size and configuration. To validate our model, we ran an exhaustive experiment with varying sort sizes for all three shuffle implementations and compared the results with the predicted values as shown in Figure 3.10.

We find that Locus's model predicts performance and cost trends pretty well, with an average error of 16.9% for run-time and 14.8% for cost. Among different sort implementations, predicting Redis-only is most accurate with an accuracy of 9.6%, then Hybrid-sort of 18.2%, and S3-only sort of 21.5%. This might due to the relatively lesser variance we see in network bandwidth to our dedicated Redis cluster as opposed to S3 which is a globally shared resource. We also notice that our prediction on average under-estimates run-time by 11%. This can be attributed to the fact that we don't model a number of other overheads such as variance in CPU time, scheduling delay etc. Overall, similar to database query optimizers, we believe that this accuracy is good enough to make coarse grained decisions about shuffle methods to use.

### 3.4.5 Big Data Benchmark

The Big Data Benchmark contains a query suite derived from production databases. We consider Query 3, which is a join query template that reads in 123GB of input and then performs joins of various sizes. We evaluate Locus to see how it performs as join size changes. We configure Locus to use 160 workers, Spark to use 5 `c3.xlarge`, and

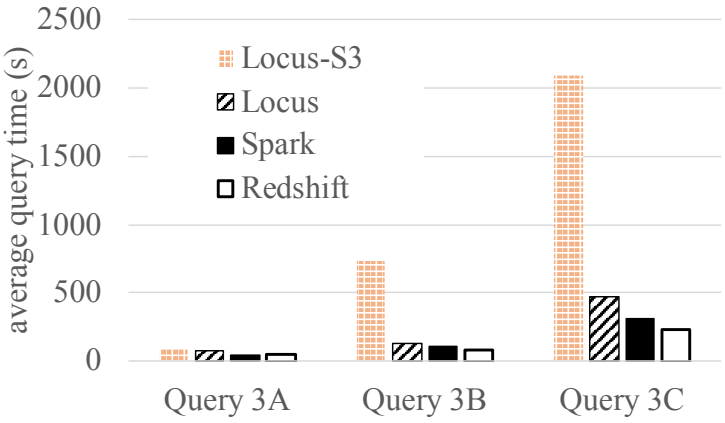



Figure 3.14: Big Data Benchmark.

Redshift to use `5ds2.8xlarge`, all totalling 160 cores. Figure 3.14 shows that even without the benefit of elasticity, Locus performance is within 1.75× to Apache Spark and 2.02× to Redshift across all join sizes. The gap is similar to what we observe in Section 3.4.1. We also see that using a default slow-storage only configuration can be up to 4× slower.

## 3.5 Related Works

**Shuffle Optimizations**: As a critical component in almost all data analytics system, shuffle has always been a venue for performance optimization. This is exemplified by Google providing a separate service just for shuffle [45]. While most of its technical details are unknown, the Google Cloud Shuffle service shares the same idea as Locus in that it uses elastic compute resources to perform shuffle externally. Modern analytics systems like Hadoop [101] or Spark [117] often provide multiple communication primitives and sort implementations. Unfortunately, they do not perform well in a serverless setting, as shown previously. There are many conventional wisdom on how to optimize cache performance [47], we explore a similar problem in the cloud context. Our hybrid sort extends on the classic idea of mergesort (see survey [33]) and cache-sensitive external sort [76, 96] to do joint optimization on the cache size and sort algorithm. There are also orthogonal works that focus on the network layer. For example, CoFlow [23] and Varys [24] proposed coordinated flow scheduling algorithms to achieve better last flow completion time. For join operations in databases, Locus relies on existing query compilers to generate shuffle plans. Compiling the optimal join algorithm for a query is an extensively studied area in databases [25], and we plan to integrate our shuffle characteristics with database optimizers in the future.

**Serverless Frameworks**: The accelerated shift to serverless has brought innovations to SQL processing [16, 10, 44, 91], general computing platforms (OpenLambda [51], AWS Lambda, Google Cloud Functions, Azure Functions, etc.), as well as emerging general computation frameworks [11, 36] in the last two years. These frameworks are architected in different ways: AWS-Lambda [11] provides a schema to compose MapReduce queries with existing AWS services; ExCamera [36] implemented a state machine in serverless tasks to achieve fine-grained control; Prior work [32] has also looked at exploiting the usability aspects to provide a seamless interface for scaling unmodified Python code.

**Database Cost Modeling**: There has been extensive study in the database literature on building cost-models for systems with multi-tier storage hierarchy [66, 64] and on targeting systems that are bottlenecked on memory access [17]. Our cost modeling shares a similar framework but examines costs in a cloud setting. The idea of dynamically allocating virtual storage resource, especially fast cache for performance improvement can also be found in database literature [103]. Finally, our work builds on existing techniques that estimate workload statistics such as partition size, cardinality, and data skew [67].

## 3.6 Summary

With the shift to serverless computing, there have been a number of proposals to develop general computing frameworks on serverless infrastructure. However, due to resource limits and performance variations that are inherent to the serverless model, it is challenging to efficiently execute complex workloads that involve communication across functions. In this chapter, we show that using a mixture of slow but cheap storage with fast but expensive storage is necessary to achieve a good cost-performance trade-off. We presents Locus, an analytics system that uses performance modeling for shuffle operations executed on serverless architectures. Our evaluation shows that the model used in Locus is accurate and that it can achieve comparable performance to running Apache Spark on a provisioned cluster, and within 2 $\times$ slower compared to Redshift. We believe the performance gap can be improved in the future, and meanwhile Locus can be preferred as it requires no provisioning of clusters.

# Chapter 4

# Fair Sharing for Elastic Memory

Memory caches continue to be a critical component to many systems. In recent years, there has been larger amounts of data into main memory, especially in shared environments such as the cloud. The nature of such environments requires resource allocations to provide both performance isolation for multiple users/applications and high utilization for the systems. We study the problem of fair allocation of memory cache for multiple users with shared files. We find that, surprisingly, no memory allocation policy can provide all three desirable properties (isolation-guarantee, strategy-proofness and Pareto-efficiency) that are typically achievable by other types of resources, e.g., CPU or network. We also show that there exist policies that achieve any two of the three properties. We find that the only way to achieve both isolation-guarantee and strategy-proofness is through *blocking*, which we efficiently adapt in a new policy called FairRide. We implement FairRide in a popular memory-centric storage system using an efficient form of *blocking*, named as *expected delaying*, and demonstrate that FairRide can lead to better cache efficiency (2.6× over isolated caches) and fairness in many scenarios.

## 4.1 Background

Most of today's cache systems are oblivious to the entities (users) that access data: CPU caches do not care which thread accesses data, web caches do not care which client reads a web page, and in-memory based systems such as Spark [119] do not care which user reads a file. Instead, these systems aim to maximize system efficiency (e.g., maximize hit rate) and as a result favor users that contribute more to improve efficiency (e.g., users accessing data at a higher rate) at the expense of the other users.

To illustrate the unfairness of these cache systems, consider a typical setup of a hosted service, as shown in Figure 4.1. We setup multiple hosted sites, all sharing a single Memcached [71] caching system to speed up the access to a back-end database. Assume the loads of $A$ and $B$ are initially the same. In this case, as expected, the mean request latencies for the two sites are roughly the same (see left bars in Figure 4.2). Next, assume that the load of site $A$ increases significantly. Despite the fact that $B$'s load remains constant, the mean latency of its requests increases significantly (2.9×) and the latency for $A$'s requests surprisingly drops! Thus, an increase in $A$'s load improves the performance of $A$, but degrades the performance of $B$. This is because $A$ accesses the data more frequently, and in response the cache system starts loading more results from $A$ while evicting $B$'s results.

While the example is based on synthetic web workload, this problem is very real, as demonstrated by the many questions posted on technical forums [57], on how to achieve resource isolation across multiple sites when using either Redis [88] or Memcached [71]. It turns out that none of the two popular caching systems provide any guarantee for performance isolation. This includes customized distributions from dominant cloud service providers, such as Amazon ElastiCache [2] and Microsoft Azure Redis Cache [12]. As we will show in Section 4.6, for such caching systems, it is easy for a strategic user to improve her performance and hurt others (with 2.9× performance gap) by making spurious access to files.

To provide performance isolation, the default answer in the context of cloud cache services today is to setup a separate caching instance per user or per application. This goes against consolidation and comes at a high cost. Moreover, cache isolation will eliminate the possibility of *sharing cached files*, which makes isolation even more expensive as there is a growing percentage of files to be shared. We studied a production HDFS log from a Internet company and observed 31.4% of files are shared by at least two users/applications. The shared files also tend to be more frequently accessed compared to non-shared files, e.g., looking at the 10% most accessed files, shared files account for as much as 53% of the accesses. The percentage of sharing can go even higher pair-wise: 22% of the users have at least 50% of their files accessed by another user. Assuming files are of equal sizes, we would need at least 31.4% more space if we assign isolated instances for each user, and even more cost on additional cache as the percentage of shared files in the working set is even larger.

Going back to the example in Figure 4.1, one possible strategy for $B$ to reclaim some of

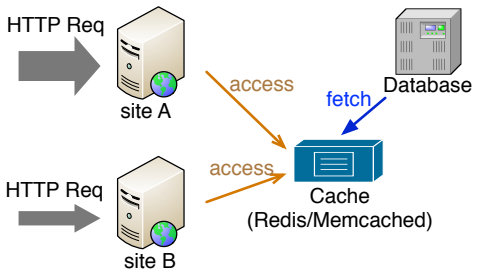



Figure 4.1: Typical cache setup for web servers.

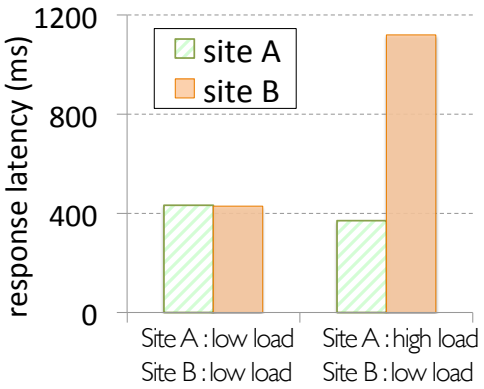



Figure 4.2: Site B suffers high latency because unfair cache sharing.

its cache back would be to artificially increase its access rate. While this strategy may help *B* to improve its performance, it can lead to worse performance overall (e.g., lower aggregate hit rate). Worse yet, site *A* may decide to do the same: artificially increase its access rate. As we will show in this chapter, this may lead to everyone losing out, i.e., everyone getting worse performance than when acting truthfully. Thus an allocation policy such as LRU is not strategy proof, as it does incentivize a site to misbehave to improve its performance. Furthermore, like with prisoner's dilemma, sites are incentivized to misbehave even if this leads to worse performance for everyone.

While in theory users might be incentivized to misbehave, a natural question is whether they are actually doing so in practice. The answer is "yes", with many real-world examples being reported in the literature. Previous works on cluster management [111] and job scheduling [42] have reported that users lie about their resource demands to game the system. Similarly, in peer-to-peer systems, "free-riding" is a well known and wide spread problem. In an effort to save bandwidth and storage, "free-riders" throttle their uplink bandwidth and remove files no longer needed, which leads to decreased overall performance [85]. We will show in Section 4.2 that shared files can easily lead to free-riding in cache allocation. Finally, as mentioned above, cheating in the case of caching is as easy as artificially increasing the access rate, for example, by running an infinite loop that accesses the data of interest, or just by making some random access. While some forms of cheating do incur certain cost or overhead (e.g., CPU cycles, access quota), the overhead is outweighed by the benefits obtained. On the one hand, a strategic user does not need many spurious accesses for effective cheating, as we will show in Section 4.6. If a caching system provides interfaces for users to specify file priorities or evict files in the system, cheating would be even simpler. On the other hand, many applications' performances are bottle-necked at I/O, and trading off some CPU cycles for better cache performance is worthwhile.

In summary, we argue that any caching allocation policy should provide the following three properties: (1) *isolation-guarantee* which subsumes performance isolation (i.e., a user will not be worse off than under static isolation), (2) *strategy-proofness* which ensures that a user cannot improve her performance and hurt others by lying or misbehaving, and (3) *Pareto efficiency* which ensures that resources are fully utilized.

## 4.2 Pareto Efficiency vs. Strategy Proofness

In this section we show that—under the assumption that the isolation-guarantee property holds—there is a strong trade-off between Pareto efficiency and strategy-proofness, that is, it is not possible to simultaneously achieve both in a caching system where files (pages) can be shared across users.

**Model:** To illustrate the above point, in the remainder of this section we consider a simple model where multiple users access a set of files. For generality we assume each user lets the cache system know the *priorities* in which her files can be evicted, either by explicitly specifying the priorities on the files or based on a given policy, such as LFU or LRU. For simplicity, in all examples, we assume that all files are of unit size.

**Utility:** We define each user's utility function as the *expected cache hit rate*. Given a cache allocation, it's easy to calculate a user's expected hit rate by just summing up her access frequencies of all the files cached in memory.

### 4.2.1 Max-min Fairness

One of the most popular solutions to achieve efficient resource utilization while still providing isolation is *max-min fairness*.

```
    FUNC u.access(f)   // user u accessing file f
 1: if (f ∈ Cache.fileSet) then
 2:     return CACHED_DATA;
 3: else
 4:     return CACHE_MISS;
 5: end if

    FUNC cache(u, f)   // cache file f for user u
 6: while (Cache.availableSize < f.size) do
 7:     u1 = users.getUserWithLargestAlloc();
 8:     f1 = u1.getFileToEvict();
 9:     if (u1 == u and
10:       u.getPriority(f1) > u.getPriority(f)) then
11:       return CACHE_ABORT;
12:     end if
13:     Cache.fileSet.remove(f1);
14:     Cache.availableSize += f1.size;
15:     u.allocSize -= f1.size;
16: end while
17: Cache.fileSet.add(f)
18: Cache.availableSize -= f.size;
19: u.allocSize += f.size;
20: return CACHE_SUCCEED;
```

Algorithm 1: Pseudocode for accessing and cahing a file under max-min fairness.

In a nutshell, max-min fairness aims to maximize the minimum allocation across all users. Max-min fairness can be easily implemented in our model by evicting from the user with the largest cache allocation, as shown in Algorithm 1. When a user accesses a file, $f$, the system checks whether there is enough space available to cache it. If not, it repeatedly evicts the files of the users who have the largest cache allocation to make enough room for $f$. Note that the user from which we evict a file can be the same as the user who is accessing file $f$, and it is possible for $f$ to not be actually cached. The latter happens when $f$ has a lower caching $priority$ than any of the other user's files that are already cached. At line 10 from Algorithm 1, the $user.getPriority()$ is called to obtain $priority$. Note caching $priority$ depends on the eviction policy. In the case of LFU, $priority$ represents file's access frequency, while in the case of LRU it can represent the inverse of the time interval since it has been accessed. Similar to access frequency, $priority$ need not to be static, but rather reflects an eviction policy's instantaneous preference.

If all users have enough demand, max-min fairness ensures that each user will get an equal amount of cache, and max-min fairness reduces to static isolation. However, if one

or more users do not use their entire share, the unused capacity is distributed across the other users.

### 4.2.2 Shared Files

So far we have implicitly assumed that each user accesses different files. However, in practice multiple users may share the same files. For example, different users or applications can share the same libraries, input files and intermediate datasets, or database views.

The ability to share the same allocation across multiple users is a key difference between caching and traditional environments, such as CPU and communication bandwidth, in which max-min fairness has been successfully applied so far. With CPU and communication bandwidth, only a single user can access the resource that was allocated to her: a CPU time slice can be only used by a single process at a time, and a communication link can be used to send a single packet of a single flow at a given time.

A natural solution to account for shared files is to "charge" each user with a *fraction* of the shared file's size. In particular, if a file is accessed by $k$ users, and that file is cached, each user will be charged with $1/k$ of the size of that file. Let $f_{i,j}$ denote file $j$ cached on behalf of user $i$, and let $k_j$ denote the number of users that have requested the caching of file $j$. Then, the total cache size *allocated* to user $i$, $alloc_i$, is computed as

$$alloc_i = \sum_j \frac{\text{size}(f_{i,j})}{k_j}. \tag{4.1}$$

Consider a cache that can hold 6 files, and assume three users. User 1 accesses files $A, B, C, \ldots$ user 2 accesses files $A, B, D, \ldots$, and user 3 accesses files $F, G, \ldots$. Assuming that each file is of unit size, the following set of cached files represent a valid max-min allocation: $A, B, C, D, F$, and $G$, respectively. Note that since files $A$ and $B$ are shared by the first two users, each of these users is only charged with half of the file size. In particular, the cache allocation of user 1 is computed as $\text{size}(A)/2 + \text{size}(B)/2 + \text{size}(C) = 1/2 + 1/2 + 1 = 2$. The allocation of user 2 is computed in a similar manner, while allocation of user 3 is simply computed as $\text{size}(F) + \text{size}(G) = 2$. The important point to note here is that while each user has been allocated the same amount of cache as computed by Eq. 4.1, users 1 and 2 get three files cached (as they get the benefit of sharing two of them), while user 3 gets only two.

### 4.2.3 Cheating

While max-min fairness is strategy-proof when users access different files, this is no longer the case when files are shared. There are two types of cheating that could break strategy-proofness: (1) Intuitively, when files are shared, a user can "free ride" files that have been already cached by other users. (2) A thrifty user can choose to cache files that are shared by more users, as such files are more economic due to cost-sharing.

**Free-riding** To illustrate "free riding", consider two users: user 1 accesses files $A$ and $B$, and user 2 accesses files $A$ and $C$. Assume size of cache is 2, and that we can cache

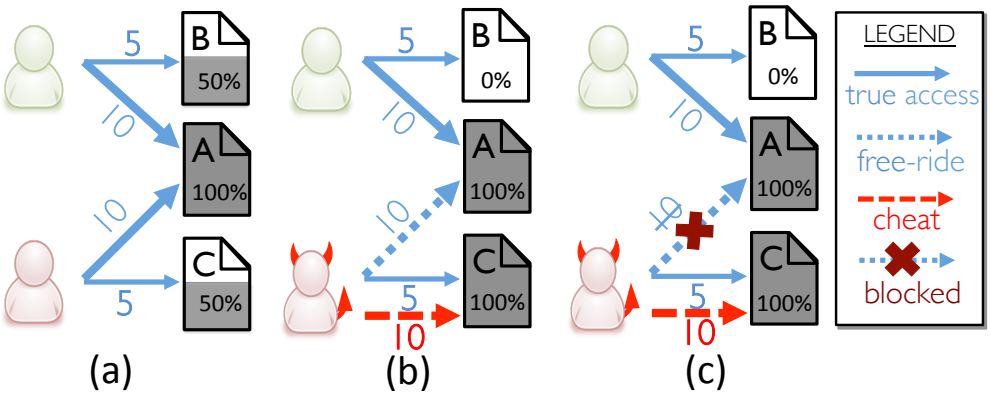



Figure 4.3: Example with 2 users, 3 files and total cache size of 2. Numbers represent access frequencies. (a). Allocation under *max-min fairness*; (b). Allocation under *max-min fairness* when second user makes spurious access (red line) to file $C$; (c). Blocking free-riding access (blue dotted line).

a fraction of a file. Next, assume that every user uses LFU replacement policy and that both users access $A$ much more frequently than the other files. As a result, the system will cache file $A$ and "charge" each user by 1/2. In addition, each user will get half of their other files in the cache, *i.e.*, half of file $B$ for user 1, and file $B$ for user 2, as shown in Figure 4.3(a). Each user gets a cache hit rate of $5 \times 0.5 + 10 = 12.5$[1] hits/sec.

Now assume user 2 cheats by spuriously accessing file $C$ to artificially increase its access rate such that to exceed $A$'s access rate (Figure 4.3(b)), effectively sets the $priority$ of $C$ higher than $B$. Since now $C$ has the highest access rate for user 2, while $A$ remains the most accessed file of user 1, the system will cache $A$ for user 1 and $C$ for user 2, respectively. The problem is that user 2 will still be able to benefit from accessing file $A$, which has already been cached by user 1. At the end, user 1 gets 10 hits/sec, and user 2 gets 15 hits/sec. In this way, user 2 free-rides on user 1's file $A$.

**Thrifty-cheating** To explain the kind of cheating where a user carefully calculates cost-benefits and then changes file priorities accordingly, we first define cost/(hit/sec) as the amount of budget cost a user pays to get 1 hit/sec access rate for a unit file. To optimize over the utility, which is defined as the total hit rate, a user's optimal strategy is not to cache the files that one has highest access frequencies, but the ones with lowest cost/(hit/sec). Compare a file of 100MB, shared by 2 users and another file of 100MB, shared by 5 users. Even though a user access the former 10 times/sec and the latter only 8 times/sec, it is overall economic to cache the second file (comparing 5MB/(hit/sec) vs. 2.5MB/(hit/sec)).

The consequence of "thrift-cheating", however, is more complicated. As it might appear to improve user and system performance at first glance, it doesn't lead to an

[1]When half of a file is in cache, half of the page-level accesses to the file will result in cache miss. Numerically, it is the equal to missing the entire file 50% of the time. So hit rate is calculated as access rate multiplied by percentage cached.

equilibrium where all users are content about their allocations. This can cause users to constantly game the system which leads to a worse outcome.

In the above examples we have shown that due to another user cheating, one can experience utility loss. A natural question to ask is, how bad could it be? i.e. What is the upper bound a user can lose when being cheated? By construction, one can show that for two-user cases, a user can lose up to 50% of cache/hit rate when all her files are shared and "free ridden" by the other strategic user. As the free-rider evades charges of shared files, the honest user double pays. This can be extended to a more general case with $n$ ($n > 2$) users, where loss can increase linearly with the number of cheating users. Suppose that cached files are shared by $n$ users, each user pays $\frac{1}{n}$ of the file sizes. If $n-1$ strategic users decide to cache other files, the only honest user left has to pay the total cost. In turn, the honest user has to evict at most ($\frac{n-1}{n}$) of her files to maintain the same budget.

It is also worth mentioning that for many applications, moderate or even minor cache loss can result in drastic performance drop. For example, in many file systems with overall high cache hit ratio, the effective I/O latency with caching could be approximated as $T_{IO} = Ratio_{miss}Latency_{miss}$. A slight difference in the cache hit ratio, e.g. from 99.7% to 99.4%, means 2× I/O average latency drop! This indeed necessitates strategy-proofness in cache policies.

### 4.2.4 Blocking Access to Avoid Cheating

At the heart of providing strategy-proofness is this question of how free-riding can be prevented. In the previous example, user 2 was incentivized to cheat because she was able to access the cached shared files regardless her access patterns. Intuitively, if user 2 is blocked from accessing files that she tries to free-ride, she will be dis-incentivized to cheat.

Applying blocking to our previous example, user 2 will not be allowed to access $A$, despite the fact that user 1 has already cached $A$ (Figure 4.3(c)). The system blocks user 2 but not user 1 because user 1 is the sole person who pays the cache. As a result, user 2 gets only 1 cache size with a less important file $C$.

As we will show in Section 4.4 this simple scheme is strategy-proof. On the other hand, this scheme is unfortunately not Pareto efficient by definition, as the performance (utility) of user 2 can be improved without hurting user 1 by simply letting user 2 access file $A$.

Furthermore, note that it is not necessary to have a user cheating to arrive at the allocation in Figure 4.3. Indeed, user 2 can legitimately access file $C$ at a much higher rate than $A$, In this case, we get the same allocation—file $A$ is cached on behalf of user 1 and file $C$ is cached on behalf of user 2—with no user cheating. Blocking in this case will reduce the system utilization by punishing a well-behaved user.

Unfortunately, the cache system cannot differentiate between a cheating and a well-behaved user, so it is not possible to avoid the decrease in the utilization and thus the violation of Pareto efficiency, even when every user in the system is well-behaved.

Thus, in the presence of shared files, with max-min fairness allocation we can achieve either Pareto efficiency or strategy-proofness, but not both. In addition, we can trade between strategy-proofness and Pareto efficiency by blocking a user from accessing a shared file if that file is not in the user's cached set of files, even though that file might have been cached by other users.

In Section 4.4, we will show that this trade-off is more general. In particular, we show that in the presence of file sharing there is no caching allocation policy that can achieve more than two out of the three desirable properties: isolation-guarantee, strategy-proofness, and Pareto efficiency.

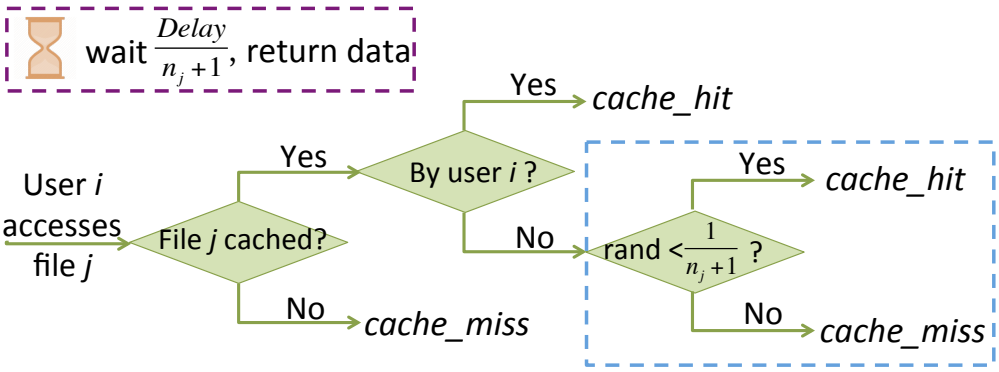



Figure 4.4: With FairRide, a user might be blocked to access a cached copy of file if the user does not pay the storage cost. The blue box shows how this can be achieved with *probabilistic blocking*. In system implementation, we replace the blue box with the purple box, where we instead delay the data response.

## 4.3 FairRide

In this section, we describe FairRide, a caching policy that extends max-min fairness with *probabilistic blocking*. Different from max-min fairness, FairRide provides isolation-guarantee and strategy-proofness at the expense of Pareto-efficiency. We use *expected delaying* to implement the conceptual model of *probabilistic blocking*, due to several system considerations.

Figure 4.4 shows the control logic for a user $i$ accessing file $j$ under FairRide. We will compare it with the pseudo-code of max-min fairness, Algorithm 1. In max-min, a user $i$ can directly access a cached file $j$, as long as $j$ is cached in memory. While with FairRide, there is an chance that the user might get blocked for accessing the cached copy. This is key to making FairRide strategy-proof and the *only* difference with max-min fairness, which we prove in Section 4.4. The chance of blocking is not an arbitrary probability, but is set at $\frac{1}{n_j+1}$, where $n_j$ is the number of other users caching the file. We will prove in Section 4.4 that this is the only and minimal blocking probability setting that will make a FairRide strategy-proof.

Consider again the example in Figure 4.3. If user 2 cheats and makes spurious access to file $C$, file $A$ will be cached on behalf of user 1. In that case, FairRide recognizes user 2 as a non-owner of the file, and user 2 has $\frac{1}{2}$ chance to access directly from the cache. So user 2's total expected hit rate becomes $5+10\times\frac{1}{2}=10$, which is worse than 12.5 before without cheating. In this way, FairRide discourages cheating and makes the policy strategy-proof.

### 4.3.1 Expected Delaying

In real systems, *probabilistic blocking* could not thoroughly solve the problem of cheating, as now a strategic user can make even more accesses in hope that one of the accesses is not blocked. For example, if a user is blocked with a probability of $\frac{1}{2}$, he can make

three accesses so to reduce the likelihood of being blocked to $\frac{1}{8}$. In addition, *blocking* itself is not an ideal way to implement in a system as it further incurs unnecessary I/O operations (disk, network) for blocked users. To address this problem, we introduce *expected delaying* to approximate the expected effect of *probabilistic blocking*. When a user tries to access an in-memory file that is cached by other users, the system delays the data response with certain wait duration. The wait time should be set as the expected delay a user would experience if she's probabilistically blocked by the system. In this way, it is impossible to get around the delaying effect, and the system does not have to issue additional I/O operations. The theoretically equivalent wait time could be calculated as $t_{wait} = Delay_{mem} \times (1 - p_{block}) + Delay_{disk,network} \times p_{block}$, where $p_{block}$ is the blocking probability as described above, and $Delay_x$ being the access latency of medium $x$. As memory access latency is already incurred during data read time, we simply set the wait time to be $Delay_{disk,newtwork} \times p_{block}$. We will detail how we measure the secondary storage delay in Section 4.5.

## 4.4 Analysis

In this section, we prove that the general trade-off between the three properties is fundamental with existence of file sharing. Next in Section 4.4.2, we also show by proof that FairRide indeed achieves strategy-proof and isolation-guarantee, and that FairRide uses most efficient blocking probability to achieve strategy-proofness.

### 4.4.1 The SIP theorem

We state the following **SIP theorem**: *With file sharing, no cache allocation policy can satisfy all three following properties: strategy-proofness (S), isolation-guarantee (I) and Pareto-efficiency (P).*

**Proof of the SIP theorem**
The three properties are defined as in Section 1.4, and we use total hit rate as the performance metric. Reusing the example setup in Figure 4.3(a), we now examine a *general* policy $P$. The only assumption of $P$ is that $P$ satisfies isolation-guarantee and Pareto-efficiency, and we shall prove that such policy $P$ must not be strategy-proof, i.e. a user can cheat to improve under $P$. We start with the case when no user cheats for Figure 4.3(a). Let $y_1$, $y_2$ be user 1 and 2's total hit rate:

$$y_1 = 10x_A + 5x_B \tag{4.2}$$

$$y_2 = 10x_A + 5x_C \tag{4.3}$$

Where $x_A$, $x_B$, $x_C$ are fractions of the each file $A, B, C$ cached in memory.[2] Because $x_A + x_B + x_C = 2$, and $y_1 + y_2 = 15x_A + 5(x_A + x_B + 5x_c)$, it's impossible for $y_1 + y_2 > 25$, or, for both $y_1$ and $y_2$ to be greater than 12.5. As the two users have symmetric access patterns, we assume $y_2 < 12.5$ without loss of generality.

Now if user 2 cheats and increases her access rate of file $C$ to 30, we can prove that she can get a total rate of 13.3, or $y_2 > 13.3$. This is partly because the system has to satisfy a new isolation guarantee:

$$y_2' = 10x_A + 30x_C > 30 \tag{4.4}$$

It must hold that $x_C > \frac{2}{3}$, because $x_A \leq 1$. Also, because $x_C \leq 1$ and $x_A + x_B + x_C = 2$, we have $x_A + x_B \geq 1$ to achieve Pareto-efficiency. For the same reason, $x_A = 1$ is also necessary as it's strictly better to cache file $A$ over file $B$ for both users. Plugging $x_A = 1, x_C > \frac{2}{3}$ back to user 2's actual hit rate calculation (Equation 4.3), we get $y_2 > 13.3$.

So far, we have found a cheating strategy for a user 2 to improve her cache performance and hurt the other user. This is done under a general policy $P$ that assumes only isolation-guarantee and Pareto-efficiency but nothing else. Therefore, we can conclude that any policy $P$ that satisfies the two properties cannot achieve strategy-proofness. In other

[2]We use fractions only for simplifying the proof. The theorem holds when we can only cache a file/block in its entirety.

words, no policy can achieve all three properties simultaneously. This ends the proof for the SIP theorem.

### 4.4.2 FairRide Properties

We now examine FairRide (as described in Section 4.3) against three properties.

**Theorem** *FairRide achieves isolation-guarantee.*

**Proof** Even if FairRide does complete blocking, in which each user gets strictly less memory cache, the amount of cache a user accesses is: $Cache_{total} = \sum_j size(file_j)$, $j$ for all the files the user caches. Because FairRide splits the charges of shared files across all users, a user's allocation budget is spent up by: $Alloc = \sum_j \frac{size(file_j)}{n_j}$, with $n_j$ being the number of users sharing $file_j$. Combining the two equations we can easily derive that $Cache_{total} > Alloc$. As $Alloc$ is also what a user can get in *isolation*, we can conclude that the amount of memory a user can access is always bigger than *isolation*. Likewise, we can prove the total hit rate user gets with FairRide is greater than *isolation*.

**Theorem** *FairRide is strategy-proof.*

**Proof** We will sketch the proof using cost-benefit analysis, following the line of reasoning in Section 4.2.3. With *probabilistic blocking*, a user $i$ can access a file $j$ without caching it with a probability of $\frac{n_j}{n_j+1}$. This means that the benefit resulted from caching is the $increased\_rate$, equal to $freq_{ij}\frac{1}{n_j+1}$. The cost is $\frac{1}{n_j+1}$ for the joining user, with $n_j$ other users already caching it. Dividing the two, the benefit-cost ratio is equal to $freq_{ij}$, user $i$'s access frequency of file $j$. As a user is incentivized to cache files based on the descending order of benefit-cost ratio, this results in caching files based on actual access frequencies, rather than cheating. In other words, FairRide is incentive-compatible and allows users to perform truth-telling.

**Theorem** *FairRide's uses lower-bound blocking probabilities for achieving strategy-proofness.*

**Proof** Suppose a user has 2 files: $f_j$,$f_k$ with access frequencies of $freq_j$ and $freq_k$. We use $p_j$ and $p_k$ to denote the corresponding blocking probabilities if the user chooses not to cache the files. Then the benefit-cost ratios for the two files are $freq_j p_j(n_j + 1)$ and $freq_k p_k(n_k + 1)$, $n_j$ and $n_k$ being the numbers of other users already caching the files. For the user to be truth-telling for whatever $freq_j$, $freq_k$, $n_j$ or $n_k$, we must have $\frac{p_j}{p_k} = \frac{n_k+1}{n_j+1}$. Now $p_j$ and $p_k$ can still be arbitrarily small or big, but note $p_j(p_k)$ must be 1 when $n_j(n_k)$ is 0, as no user is caching file $f_j(f_k)$. Putting $p_j = 1, n_j = 0$ into the equation we will have $p_k = \frac{1}{n_k+1}$. Similarly, $p_j = \frac{1}{n_j+1}$. Thus we show that FairRide's blocking probabilities are the only probabilities that can provide strategy-proofness in the general case (assuming any access frequencies and sharing situations). The only probabilities are also the lower-bound probabilities.

## 4.5 Implementation

FairRide is evaluated through both system implementation and large-scale trace simulations. We have implemented FairRide allocation policy on top of Tachyon [62], a memory-centric distributed storage system. Tachyon can be used as a caching system and it supports in-memory data sharing across different cluster computation frameworks or applications, e.g. multiple Hadoop Mapreduce [8] or Spark [119] applications.

**Users and Shares** Each application running on top of Tachyon with FairRide allocation has a FairRide client ID. Shares for each user can be configured. When shares are changed during system uptime, cache allocation is re-allocated over time, piece by piece, by evicting files from the user who uses most atop of her share, i.e., $argmax_i(Alloc_i - Capacity * Share_i)$, thus converging to the configured shares eventually.

**Pluggable Policy** Because FairRide obeys each user's individual caching preferences, it can apply a two-level cache replacement mechanism. It first picks the user who occupies the most cache in the system, and then finds the least preferred file from that user to evict. This naturally enables "pluggable policy", allowing each user to pick a replacement policy best fit for her workload. Note this would not be possible for some global policies such as global LRU. A user's more frequently accessed file could be evicted by a less frequently accessed file just because the first file's aggregate frequency across all users is lower than the second file. We've implemented "pluggable policy" in the system and expose a simple API for applications to pick best replacement policy.

```
Client.setCachePolicy(Policy.LRU)
Client.setCachePolicy(Policy.LFU)
Client.pinFile(fileId)
```

Currently, our implementation of FairRide supports LRU (Least-Recently-Used) and LFU (Least-Frequently-Used), as well as policies that are more suited for data-parallel analytics workloads, e.g. LIFE or LFU-F that preserves all-or-nothing properties for cached files [6]. Another feature FairRide supports is "pinned files". Through a `pinfile(fileId)` API, a user can override the replacement policy and prioritize specified files.

**Delaying** The key to strategy-proofness in implementing FairRide is to emulate *probabilistic blocking* by *delaying* the read of a file which a user didn't cache before. Thus the amount of wait time has to approximate the wait time as if the file is not cached, for any type of read. We implement *delaying* by simply sleeping the thread before giving a data buffer to the Tachyon client. The delay time is calculated by $\frac{size(buffer)}{BW_{disk}}$, with $BW_{disk}$ being the pre-measured disk bandwidth on the node, and $size(buffer)$ being the size of the data buffer sent to the client. The measured bandwidth is likely an over-estimate of run-time disk bandwidth due to I/O contention when system is in operation. This causes shorter

delay, higher efficiency, and less strategy-proofness, though a strategic user should gain very little from this over-estimate.

**Node-based Policy Enforcement**

Tachyon is a distributed system comprised of multiple worker nodes. We enforce allocation policies independently at each node. This means that data is always cached locally at the node when being read, and that when the node is full, we evict from the user who uses up most memory on that node. This scheme allows a node to select an evicting user and perform cache replacement without any global coordination.

The lack of global coordination can incur some efficiency penalty, as a user is only guaranteed to get at least $1/n$-th of memory on each node, but not necessarily $1/n$-th of total memory across the cluster. This happens when users have access skew across nodes. To give an example, suppose a cluster of two nodes, each with 40GB memory. One user has 30GB frequently accessed data on node 1 and 10GB on node 2, and another user has 10GB frequently accessed data on node 1 and 30GB on node 2. Allocating 30GB on node 1 and 10GB on node 2 to the first user will outperform a 20 to 20 even allocation on each node, in terms of hit ratio for both users. Note that such allocation is still fair globally – each user gets 40GB memory in total. Our evaluation results in Section 4.6.6 will show that node-based scheme is within 3%~4% compared to global fairness, because of the self-balance nature of big data workloads on Tachyon.

## 4.6 Experimental Results

We evaluated FairRide using both micro- and macro-benchmarks, by running EC2 experiments on Tachyon, as well as large-scale simulations replaying production workloads. The number of users in the workloads varies from 2 to 20. We show that while non-strategy-proof policies can cause everybody worse-off by a large margin (1.9×), FairRide can prevent user starvation within 4% of global efficiency. It is 2.6× better than isolated caches in terms of job time reduction, and gives 27% higer utilization compared to max-min fairness.

We start by showing how FairRide can dis-incentivize cheating users by blocking them from accessing files that they don't cache, in Section 4.6.1. In Section 4.6.2, we compare FairRide against a number of schemes, including max-min fairness using experiments on multiple workloads: TPC-H, YCSB and a HDFS production log. Section 4.6.3 and Section 4.6.4 demonstrate FairRide's benefits with multiple users and pluggable policies. Finally, in Section 4.6.5, we use Facebook traces that are collected from a 2000-node Hadoop cluster to evaluate the performance of FairRide in large-scale clusters.

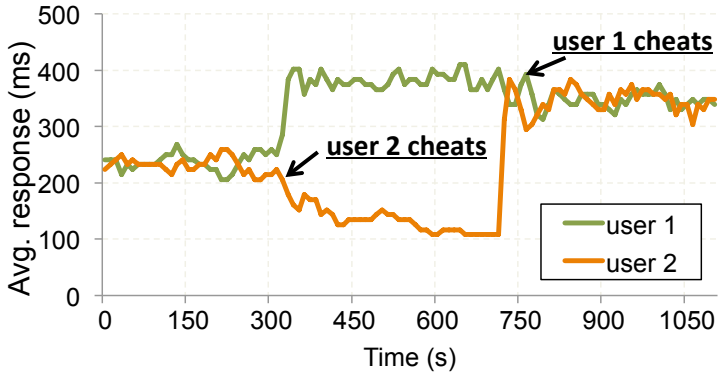
### 4.6.1 Cheating and Blocking

In this experiment, we illustrate how FairRide can prevent a user from cheating. We ran two applications on a 5-node Amazon EC2 cluster. The cluster contains one master node and four worker nodes, each configured with 32GB memory. Each application accessed 1000 data blocks (128MB each), among which 500 were shared. File access complied with Zipf distribution. We assumed users knew a priori which files are shared, and could cheat by making excessive accesses to non-shared files. We used LRU as cache replacement policy for this experiment.

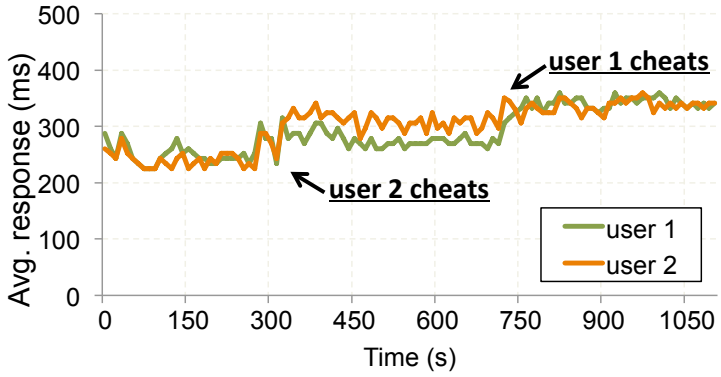
We ran the experiment under two different schemes, max-min fair allocation (Figure 4.5(a)) and FairRide (Figure 4.5(b)). Under both allocations, the two users got similar average block response time (226ms under max-min, 235ms under FairRide) at the beginning ($t < 300s$). For max-min fair allocation, when user 2 started to cheat at $t = 300s$, she managed to lower her miss ratio over time (~130ms), while user 1 got degraded performance with 380ms. At $t = 750s$, user 1 also started to cheat and both users stayed at high miss ratio (315ms). In this particular case, there was strong incentive for both the users to cheat at any point of time because cheating could always decrease the cheater's miss ratio (226ms→130ms,380ms→315ms). Unfortunately, both users get worse performance compared to not cheat all. Such a prisoner's dilemna would not happen with FairRide (Figure 4.5(a)). When user 2 cheated at $t = 300s$, her response time instead increases to 305ms . Because of this, both users would rather not cheat under FairRide and behave truthfully.

### 4.6.2 Benchmarks with Multiple Workloads

Now we evaluate FairRide by running three workloads on a EC2 cluster.




(a) Max-min fair allocation




(b) FairRide

Figure 4.5: Miss ratio for two users. At $t = 300s$, user 2 started cheating. At $t = 700s$, user 1 joined cheating.

- **TPC-H** The TPC-H benchmark [107] is a set of decision support queries based on those used by retailers such as Amazon. The queries can be separated into two main groups: a sales-oriented group and a supply-oriented group. These two groups of queries have some separate tables, but also share common tables such as those maintaining inventory records. We treated two query groups as from two independent users.

- **YCSB** The Yahoo! Cloud Serving Benchmark provides a framework and common set of workloads for evaluating the performance of key-value serving stores. We implemented a YCSB client and ran multiple YCSB workloads to evaluate FairRide. We let half of files to be shared across users.

- **Production Cluster HDFS Log** The HDFS log is collected from a production Hadoop cluster at a large Internet company. It contains detailed information such as access timestamps and access user/group information. We found that more than 30% of files are shared by at least two users.

We ran each workload under the following allocation schemes: 1) *isolation*: statically partition the memory space across users; 2) *best-case*: i.e. max-min fair allocation and assume no user cheats; 3) FairRide: our solution which uses *delaying* to prevent cheating; 4) *max-min* : max-min fair allocation with half users trying to game the system. We used LRU as the default cache replacement algorithm for all users and assumed cheating users know what files are shared.

We focus on three questions: 1) does sharing the cache improve performance significantly? (comparing performance gain over *isolation*) 2) can FairRide prevent cheating with small efficiency loss? (comparing FairRide with *best-case*) 3) does cheating degrade system performance significantly? (comparing FairRide with *max-min*).

To answer these questions, we plot the relative gain of three schemes compared to *isolation*, as shown in Figure 4.6. In general, we find sharing the cache can improve performance by 1.3~3.1×, with *best-case*. If users cheat, 15%~220% of the gain will be lost. For the HDFS workload, we also observe that cheating causes a performance drop below *isolation*. While FairRide is very close to *best-case* with 3%~13% overhead, it prevents the undesirable performance drop.

There are other interesting observations to note. First of all, the overhead of FairRide, is more noticeable in the YCSB benchmark and TPC-H than in the HDFS trace. We find that this is because the most shared files in the HDFS prodcution trace are among the top accessed files for both users. Therefore, both users would cache the shared files, resulting in less *blocking/delaying*. Secondly, cheating user benefits less in the HDFS trace, this is due to fact that the access distribution across files are highly long tailed in that trace, so that even cheating help user gain more memory, it doesn't show up significantly in terms of miss ratio. Finally, there is a varied degree of connection between miss ratio and application performance (read latency, query time), e.g., YCSB's read latency is directly linked to miss ratio change, while TPC-H's query response time is relatively stable. This is because, for the latter, a query typically consists of multiple stages of parallel tasks. As the completion time of a stage is decided by the slowest task, caching could only help when all tasks speed up. Therefore, a caching algorithm that can provide all-or-nothing caching for parallel tasks is needed to speed up query response time. We evaluated the Facebook trace with such a caching algorithm in Seciton 4.6.5.

### 4.6.3 Many Users

We want to understand how the effect of cheating relates to the number of active users in the system. In this experiment, we replay YCSB workloads with 20 users, where each pair of users have a set of shared files that they access commonly. Users can cheat by making excessive access to their private files. We increase the number of strategic users in different runs and plot the average miss ratio for both the strategic user group and the truthful user group in Figure 4.7. As expected, the miss ratio of the truthful group increases when there is a growing number of strategic users. What's interesting is that for the strategic group, the benefit they can exploit decreases as more and more users joining the group. With 12 strategic users, even the strategic group has worse performance

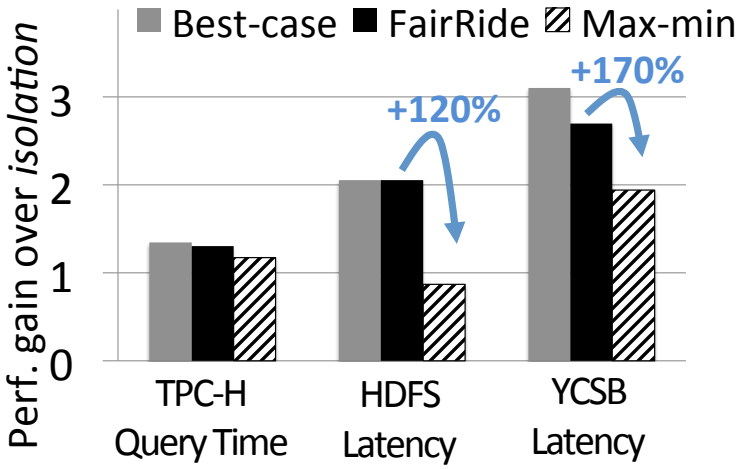



Figure 4.6: Summary of performance results for three workloads, showing the gain compared to isolated caches.

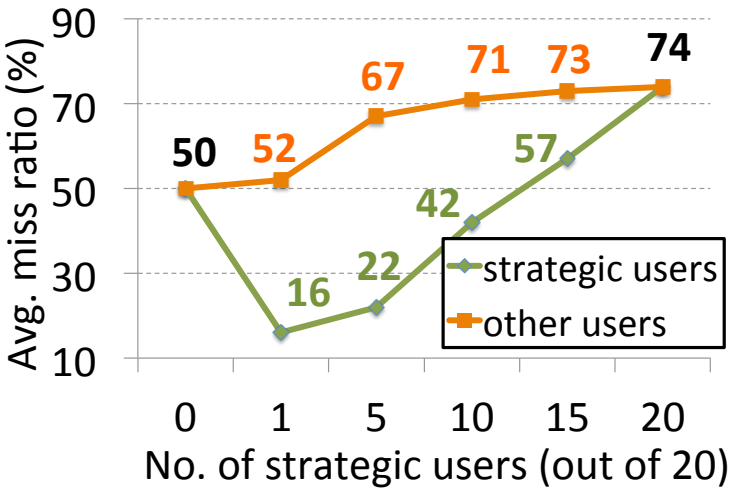



Figure 4.7: Average miss ratios of cheating users and non-cheating users, when there are multiple cheaters.

compared to the no-cheater case. Eventually both groups converge at a miss ratio of 74%.

### 4.6.4 Pluggable Policies

Next, we evaluated the benefit of allowing pluggable policies. We ran three YCSB clients concurrently with each client running a different workload. The characteristics of the three workloads are summarized below:

| User ID | Workload | Distribution | Replacement |
|---|---|---|---|
| **1** | YCSB(a) | zipfian | LFU |
| **2** | YCSB(d) | latest-most | LRU |
| **3** | YCSB(e) | scan | priority [3] |

In the experiment, each YCSB client sets up the best replacement specified in the above table with the system. We compared our system with traditional caching systems that support only configuration of one uniform replacement policy, applied to all users. We ran the system with uniform configuration three times, each time with a different policy (LRU, LFU and priority). As shown in Figure 4.8, by allowing the users to specify a best replacement policy on their own, our system is able to provide gain of the best case for each of the user among all uniform configurations.

### 4.6.5 Facebook workload

Our trace-driven simulator performed a detailed and faithful replay of a task-level trace of Hadoop jobs collected from a 2000-node cluster from Facebook during the week of October 2010. Our replay preserved read and write sizes of tasks, locations of input data as well as job characteristics of failures, stragglers.

To make the effect of caching relevant to job completion time, we also use LIFE and LFU-F from PACMan [6] as cache replacement policies. These policies performed *all-or-nothing* cache replacement for files and can improve job completion time better than LRU or LFU, as it speeds all concurrent tasks in one stage [6]. In a nutshell, LIFE evicts files based on largest-incomplete-file-first eviction, and LFU-F is based on least-accessed-incomplete-file-first. We also set each node in the cluster with 20Gb of memory so miss ratio was around 50%. The conclusion would hold for a wider range of memory size.

We adopted a more advanced model of cheating in this simulation. Instead of assuming users know what files are shared a priori, a user cheats based on the cached files she observes in the cluster. For example, for a non-blocking scheme such as max-min fairness, a user can figure out what shared files are cached by other users by continuously probing the system. She would avoid sharing the cost of those files and only cache files for her own interest.

[3] Priority replacement means keeping a fixed set of files in cache. Not the best policy here, but still better than LFU and LRU for the scan workload.

[4] Effective miss ratio. For FairRide, we count a delayed access as a "fractional" miss, with the fraction equal to the blocking probability, so we can effectively compare miss ratio between FairRide and other schemes.

Table 4.1: Summary of simulation results on reduction in job completion time, cluster efficiency improvement and hit ratio under different scheme, with no caching as baseline.

| | job time | | cluster eff. | | eff. miss%[4] | |
|---|---|---|---|---|---|---|
| | u1 | u2 | u1 | u2 | u1 | u2 |
| isolation | 17% | 15% | 23% | 22% | 68% | 72% |
| global | 54% | 29% | 55% | 35% | 42% | 60% |
| best-case | 42% | 41% | 47% | 43% | 48% | 52% |
| max-min | 30% | 43% | 35% | 47% | 63% | 46% |
| FairRide | 39% | 40% | 45% | 43% | 50% | 55% |

Caching improves overall performance of the system. Table 4.1 provides a summary of reduction in job completion time and improvement in cluster efficiency (total task runtime reduction) compared to a baseline with no caching, as well as miss ratio numbers. Similar to previous experiments, *isolation* gave lowest gains for both users and *global* improved users unevenly (compared to *best-case*). FairRide suffered minimal overhead of blocking (2% and 3% in terms of miss ratio compared to *best-case*, 4% of cluster efficiency) but could prevent cheating of user 2 that can potentially hurt user 1 by 15% in terms of miss ratio. Similar comparisons were observed in terms of job completion and cluster efficiency, FairRide can outperform *max-min* by 27% in terms of efficiency and has 2.6× more improvement over *isolation*.

Figure 4.9 also shows the reduction in job completion time across all users, plotted in median completion time (a) and 95 percentile completion time (b) respectively. FairRide preserved better overall reduction compared to *max-min*. This was due to the fact that marginal improvement of the cheating user was smaller than the performance drop of the cheated. FairRide also prevented cheating from greatly increasing the tail of job completion time (95 percentile) as the metric was more dominated by the slower user. We also show the improvement of FairRide under different cache policies in (c) and (d).

### 4.6.6 Comparing Global FairRide

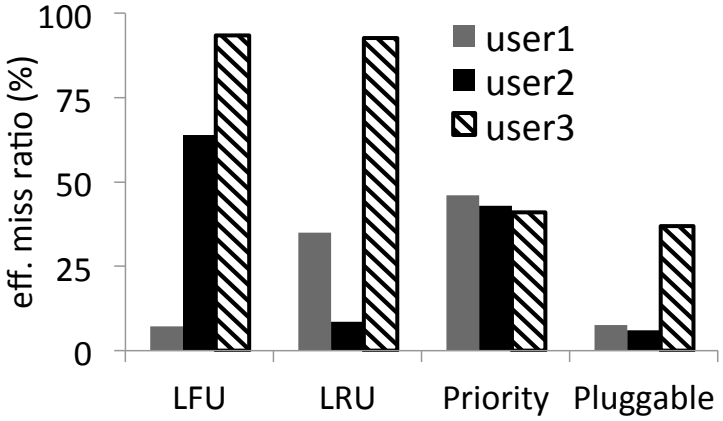
How much performance penalty does node-based FairRide suffer compared to global FairRide, if any? To answer this question, we ran another simulation with the Facebook trace to compare against two global FairRide schemes. The two global schemes both select a evicting user based on users' global usage, but differ in how they pick evicting blocks: a "naive" global scheme chooses from only blocks on that node, similar to the node-based approach, and an "optimized" global scheme chooses from any user blocks in the cluster. We use LIFE as the replacement policy for both users.

As we find out, the naive global scheme has a great performance drop (23%~25% improvement difference compared to node-based FairRide), noticeably in Table 4.2. This is due to the fact that the naive scheme is unable to allocate in favor of frequently accessing user per node. With the naive global scheme, memory allocations on each node quickly stabilizes based on initial user accesses. A user can get an unnecessarily large portion of

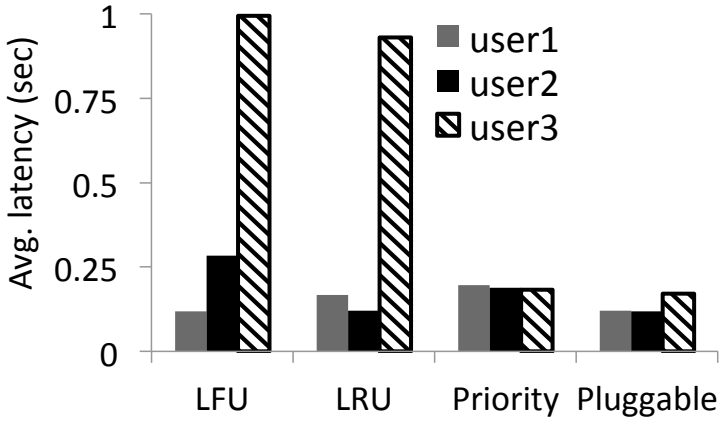
Table 4.2: Comparing against global schemes. Keep total memory size as constant while varying the number of nodes in the cluster. Showing improvement over no cache as in the reduction in median job completion time.

| **Cluster size** | **200** | **500** | **1000** |
|---|---|---|---|
| Node-based FairRide | 51% | 44% | 41% |
| Global FairRide, Naive | 25% | 21% | 17% |
| Global FairRide, Optimized | 54% | 47% | 44% |

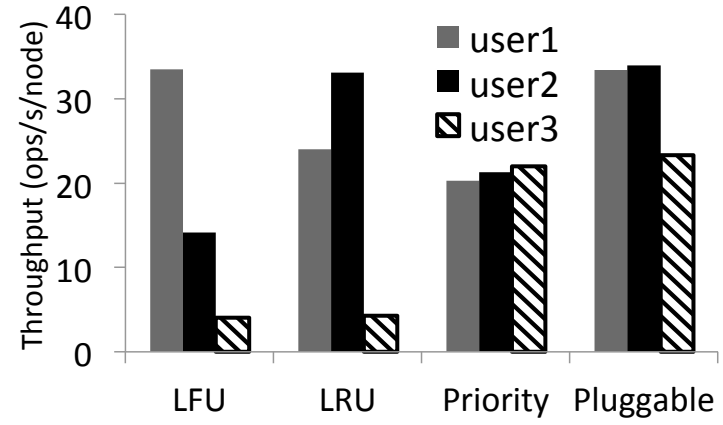
memory on a node because she accesses data earlier than the other, although her access frequency on that node is low in general. The optimized global scheme fixes this issue by allowing a user to evict least preferred data in the whole cluster and it makes sure the $1/n$-th of memory allocated must store her most preferred data. We observe an increase of average hit ratio by 24% with the optimized scheme, which reflects the access skew for the underlying data. What's interesting is that the optimized global scheme is only 3%~4% better than node-based scheme in terms of job complete time improvement. In addition to the fact data skew is not huge (considering 24% increase for hit ratio), the all-or-nothing property of data-parallel caching again comes into play. Global scheme on average increases the number of completely cached files by only 7%, and because now memory allocation is skewed across the cluster, there is an increased chance that tasks cannot be scheduled to co-locate with cached data, due to CPU slot limitation. Finally, we also observe that as the number of nodes increases (while keeping the total CPU slots and memory constant), there is a decrease in improvement in all three schemes, due to less tasks can be scheduled with cache locality.




(a) Effective Miss Ratio

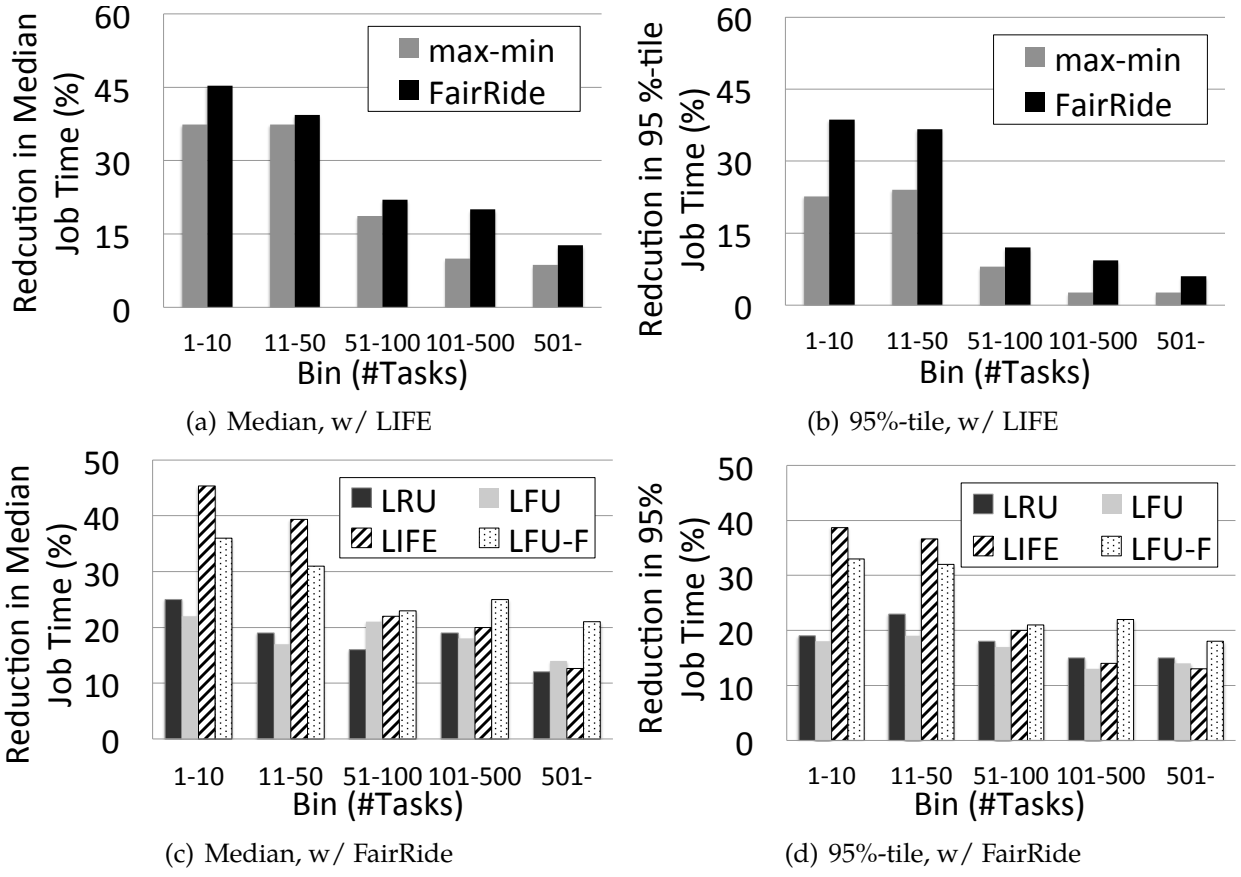



(b) Average Latency




(c) Throughput

Figure 4.8: Pluggable policies.




(a) Median, w/ LIFE

(b) 95%-tile, w/ LIFE

(c) Median, w/ FairRide

(d) 95%-tile, w/ FairRide

Figure 4.9: Overall reduction in job completion time for Facebook trace.

## 4.7 Related Works

Management of shared resources has always been been an important subject. Over the past decades, researchers and practitioners have considered the sharing of CPU [112, 20, 106, 113] and network bandwidth [68, 14, 30, 46, 99, 105], and developed a plethora of solutions to allocate and schedule these resources. The problem of cache allocation for better isolation, quality-of-service [58] or attack resilience [73] has also been studied under various contexts, including CPU cache [59], disk cache [83] and caching in storage systems [100].

One of the most popular allocation policies is *fair sharing* [27] or *max-min fairness* [65, 19]. Due to the nice properties, it has been implemented using various methods, such as round-robin, proportional resource sharing [112] and fair queuing [29], and has been extended to support multiple resource types [42] and resource constraints [41]. The key differentiator for our work from the ones mentioned above, is that we consider shared data. None of the works above identifies the impossibility of three important properties with shared files.

There are other techniques that have been studied to provide fairness and efficiency of shared cache. Prefetching of data into the cache before access, either through hints from applications [83] or predication [43], can improve the overall system efficiency. Profiling applications [59] is useful for provding application-sepcific information. We view these techniques as orthogonal to our work. Other techniques such as throttling access rate requires the system to identify good thresholds.

## 4.8 Summary

In this chapter, we study the problem of cache allocation in a multi-user environment. We show that with data sharing, it is not possible to find an allocation policy that achieves isolation-guarantee, strategy-proofness and Pareto-efficiency simultaneously. We propose a new policy called FairRide. Unlike previous policies, FairRide provides both isolation-guarantee (so a user gets better performance than on isolated cache) and strategy-proofness (so users are not incentivized to cheat), by blocking access from cheating users. We provide an efficient implementation of the FairRide system and show that in many realistic workloads, FairRide can outperform previous policies when users cheat. The two nice properties of FairRide come at the cost of Pareto-efficiency. We also show that FairRide's cost is within 4% of total efficiency in some of the production workloads, when we conservatively assume users don't cheat. Based of the appealing properties and relatively small overhead, we believe that FairRide can be a practical policy for real-world cloud environments.

# Chapter 5

# Conclusion

In a world where more and more decisions are driven by data, virtually everyone requires large processing power to analyze their data. In this world, elastic provisioning and sharing of compute and storage resources become inevitable. In the past, the cloud has taught us two great lessons to achieve this vision: resource multiplexing and resource disaggregation. We argue that both multiplexing and disaggregation have to be more aggressive to accommodate the growing needs, as exemplified by the growing popularity of serverless computing. In this dissertation, we show that one can build general analytics systems on top of a serverless infrastructure. In addition, we develop techniques that help achieve more aggressive multiplexing (by allowing fair sharing on memory) and more aggressive disaggregation (by providing a storage solution that supports serverless shuffle operations). We hope our results can be a stepping stone for more great works in the space.

# Bibliography


[1] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *OSDI*. 2016.

[2] *Amazon ElastiCache*. https://aws.amazon.com/elasticache/.

[3] G. Ananthanarayanan et al. "Reining in the Outliers in Map-Reduce Clusters using Mantri". In: *Proc. OSDI*. 2010.

[4] Ganesh Ananthanarayanan et al. "Disk-locality in datacenter computing considered irrelevant". In: *HotOS*. 2011.

[5] Ganesh Ananthanarayanan et al. "GRASS: Trimming Stragglers in Approximation Analytics". In: *NSDI*. 2014.

[6] Ganesh Ananthanarayanan et al. "PACMan: coordinated memory caching for parallel jobs". In: *NSDI'12*.

[7] Ganesh Ananthanarayanan et al. "Reining in the Outliers in Map-reduce Clusters Using Mantri". In: *OSDI*. 2010.

[8] *Apache Hadoop*. http://hadoop.apache.org/.

[9] Krste Asanovic and D Patterson. "Firebox: A hardware building block for 2020 warehouse-scale computers". In: *FAST*. 2014.

[10] *Amazon Athena*. http://aws.amazon.com/athena/.

[11] *Serverless Reference Architecture: MapReduce*. https://github.com/awslabs/lambda-refarch-mapreduce.

[12] *Azure Cache - Redis cache cloud service*. http://azure.microsoft.com/en-us/services/cache/.

[13] *Azure Blob Storage Request Limits*. https://cloud.google.com/storage/docs/request-rate.

[14] Jon CR Bennett and Hui Zhang. "WF2Q: worst-case fair weighted fair queueing". In: *INFOCOM'96*.

[15] *Big Data Benchmark*. https://amplab.cs.berkeley.edu/benchmark/.

[16] *Google BigQuery*. https://cloud.google.com/bigquery/.

[17] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access". In: *VLDB*. 1999.

[18] John Canny and Huasha Zhao. "Big data analytics with small footprint: Squaring the cloud". In: *KDD*. 2013.

[19] Zhiruo Cao and Ellen W Zegura. "Utility max-min: An application-oriented bandwidth allocation scheme". In: *INFOCOM'99*.

[20] Bogdan Caprita et al. "Group Ratio Round-Robin: O (1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems." In: *ATC'05*.

[21] Nicholas Carriero and David Gelernter. "Linda in Context". In: *CACM* 32.4 (Apr. 1989).

[22] M. Chowdhury and I. Stoica. "Coflow: A Networking Abstraction for Cluster Applications". In: *Proc. HotNets*. 2012, pp. 31–36.

[23] Mosharaf Chowdhury and Ion Stoica. "Coflow: A Networking Abstraction for Cluster Applications". In: *HotNets*. 2012.

[24] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. "Efficient Coflow Scheduling with Varys". In: *SIGCOMM*. 2014.

[25] Shumo Chu, Magdalena Balazinska, and Dan Suciu. "From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System". In: *SIGMOD*. 2015.

[26] *cloudpickle: Extended pickling support for Python objects*. `https://github.com/cloudpipe/cloudpickle`.

[27] Jon Crowcroft and Philippe Oechslin. "Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP". In: *SIGCOMM CCR, 1998* ().

[28] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proc. OSDI* (2004).

[29] A. Demers, S. Keshav, and S. Shenker. "Analysis and Simulation of a Fair Queueing Algorithm". In: *SIGCOMM'89*.

[30] Alan Demers, Srinivasan Keshav, and Scott Shenker. "Analysis and simulation of a fair queueing algorithm". In: *SIGCOMM CCR, 1989*.

[31] Matthijs Douze et al. "Evaluation of gist descriptors for web-scale image search". In: *ACM International Conference on Image and Video Retrieval*. 2009.

[32] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht. "Occupy the Cloud: Distributed Computing for the 99%". In: *SoCC*. 2017.

[33] Vladmir Estivill-Castro and Derick Wood. "A Survey of Adaptive Sorting Algorithms". In: *ACM Comput. Surv.* (1992).

[34] *IEEE P802.3ba, 40Gb/s and 100Gb/s Ethernet Task Force*. `http://www.ieee802.org/3/ba/`.

[35] Lu Fang et al. “Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs”. In: *SOSP*. 2015.

[36] Sadjad Fouladi et al. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads”. In: *NSDI*. 2017.

[37] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. “Disk-Locality in Datacenter Computing Considered Irrelevant”. In: *Proc. HotOS*. 2011.

[38] Peter X Gao et al. “Network requirements for resource disaggregation”. In: *OSDI*. 2016.

[39] *Google Cloud Storage Request Limits*. `https://docs.microsoft.com/en-us/azure/storage/common/storage-scalability-targets`.

[40] S. Ghemawat, H. Gobioff, and S.T. Leung. “The Google File System”. In: *Proc. SOSP*. 2003, pp. 29–43.

[41] Ali Ghodsi et al. “Choosy: Max-min Fair Sharing for Datacenter Jobs with Constraints”. In: EuroSys’13.

[42] Ali Ghodsi et al. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types”. In: NSDI’11.

[43] Binny S. Gill and Luis Angel D. Bathen. “AMP: Adaptive Multi-stream Prefetching in a Shared Cache”. In: *FAST’07*.

[44] *Amazon Glue*. `https://aws.amazon.com/glue/`.

[45] *Google Cloud Dataflow Shuffle*. `https://cloud.google.com/dataflow/`.

[46] Pawan Goyal, Harrick M Vin, and Haichen Chen. “Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks”. In: *SIGCOMM CCR, 1996*.

[47] Jim Gray and Goetz Graefe. “The Five-minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb”. In: *SIGMOD Rec.* (1997).

[48] Sangjin Han and Sylvia Ratnasamy. “Large-Scale Computation Not at the Cost of Expressiveness.” In: *HotOS*. 2013.

[49] Sangjin Han et al. “Network support for resource disaggregation in next-generation datacenters”. In: *HotNets*. 2013.

[50] *HDFS Caching*. `http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/`.

[51] Scott Hendrickson et al. “Serverless computation with OpenLambda”. In: *HotCloud*. 2016.

[52] Herodotos Herodotou et al. “Starfish: A Self-tuning System for Big Data Analytics.” In: *CIDR*. 2011.

[53] Simon Hettrick et al. *UK Research Software Survey 2014*. https://doi.org/10.5281/zenodo.14809. Dec. 2014. DOI: 10.5281/zenodo.14809.

[54] B. Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Proc. NSDI*. 2011.

[55] *HP The Machine: Our vision for the Future of Computing*. https://www.labs.hpe.com/the-machine.

[56] Michael Isard et al. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Proc. SOSP*. 2009, pp. 261–276.

[57] *Isolation in Memcached or Redis*. http://goo.gl/FYfrOK;http://goo.gl/iocFrt; http://goo.gl/VeJHvs.

[58] Ravi Iyer et al. "QoS Policies and Architecture for Cache/Memory in CMP Platforms". In: *SIGMETRICS'07*.

[59] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture". In: PACT'04.

[60] Horacio Andrés Lagar-Cavilla et al. "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing". In: *EuroSys*. 2009.

[61] *Using AWS Lambda with Kinesis*. http://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html.

[62] Haoyuan Li et al. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *SOCC'14*.

[63] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server." In: *OSDI*. 2014.

[64] Sherry Listgarten and Marie-Anne Neimat. "Modelling Costs for a MM-DBMS". In: *RTDB*. 1996.

[65] Qingming Ma, Peter Steenkiste, and Hui Zhang. "Routing high-bandwidth traffic in max-min fair share networks". In: *SIGCOMM CCR, 1996*.

[66] Stefan Manegold, Peter Boncz, and Martin L. Kersten. "Generic Database Cost Models for Hierarchical Memory Systems". In: *VLDB*. 2002.

[67] Michael V. Mannino, Paicheng Chu, and Thomas Sager. "Statistical Profile Estimation in Database Systems". In: *ACM Comput. Surv.* (1988).

[68] Laurent Massoulié and James Roberts. "Bandwidth sharing: objectives and algorithms". In: *INFOCOM'99*.

[69] Julian McAuley et al. "Image-based recommendations on styles and substitutes". In: *SIGIR*. 2015.

[70] Frank McSherry, Michael Isard, and Derek Gordon Murray. "Scalability! but at what COST?" In: *HotOS*. 2015.

[71] *Memcached, a distributed memory object caching system*. `http://memcached.org/`.

[72] Ivelina Momcheva and Erik Tollerud. "Software Use in Astronomy: an Informal Survey". In: *arXiv* 1507.03989 (2015).

[73] Thomas Moscibroda and Onur Mutlu. "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems". In: *USENIX Security'07*.

[74] Edmund B. Nightingale et al. "Flat Datacenter Storage". In: *OSDI*. 2012.

[75] Feng Niu et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *NIPS*. 2011.

[76] Chris Nyberg et al. "AlphaSort: A Cache-sensitive Parallel External Sort". In: *The VLDB Journal* (1995).

[77] Owen O'Malley. *TeraByte Sort on Apache Hadoop*. `http://sortbenchmark.org/YahooHadoop.pdf`.

[78] Aude Oliva and Antonio Torralba. "Modeling the shape of the scene: A holistic representation of the spatial envelope". In: *International Journal of computer vision* 42.3 (2001), pp. 145–175.

[79] *OpenWhisk*. `https://developer.ibm.com/openwhisk/`.

[80] Kay Ousterhout et al. "Making sense of performance in data analytics frameworks". In: *NSDI*. 2015, pp. 293–307.

[81] Kay Ousterhout et al. "Sparrow: distributed, low latency scheduling". In: *SOSP*. 2013.

[82] Kay Ousterhout et al. "The Case for Tiny Tasks in Compute Clusters." In: *HotOS*. 2013.

[83] R. H. Patterson et al. "Informed Prefetching and Caching". In: *SIGOPS'95* ().

[84] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications." In: *OSDI*. 2010.

[85] Michael Piatek et al. "Do Incentives Build Robustness in Bit Torrent". In: *NSDI'07*.

[86] Meikel Poess et al. "TPC-DS, Taking Decision Support Benchmarking to the Next Level". In: *SIGMOD*. 2002.

[87] Russell Power and Jinyang Li. "Piccolo: Building Fast, Distributed Programs with Partitioned Tables." In: *OSDI*. 2010.

[88] *Redis*. `http://redis.io/`.

[89] *Redis Server Side Scripting*. `https://redis.io/commands/eval`.

[90] *Redis Benchmarks*. `https://redis.io/topics/benchmarks`.

[91] *Amazon Redshift Spectrum*. `https://aws.amazon.com/redshift/spectrum/`.

[92] Xiaoqi Ren et al. “Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale”. In: *SIGCOMM* (2015).

[93] Stephen M. Rumble et al. “It’s Time for Low Latency”. In: *Proc. HotOS*. 2011.

[94] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *IJCV* 115.3 (2015), pp. 211–252.

[95] *S3 Request Limits*. `https://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html`.

[96] Betty Salzberg et al. “FastSort: A Distributed Single-input Single-output External Sort”. In: *SIGMOD*. 1990.

[97] Malte Schwarzkopf et al. “Omega: flexible, scalable schedulers for large compute clusters”. In: *Proc. EuroSys*. 2013.

[98] Colin Scott. *Latency Trends*. `http://colin-scott.github.io/blog/2012/12/24/latency-trends/`.

[99] Madhavapeddi Shreedhar and George Varghese. “Efficient fair queuing using deficit round-robin”. In: *TON’96* ().

[100] David Shue, Michael J. Freedman, and Anees Shaikh. “Performance Isolation and Fairness for Multi-Tenant Cloud Storage”. In: *OSDI’12*.

[101] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *Mass storage systems and technologies (MSST)*. 2010.

[102] *Sort Benchmark*. `http://sortbenchmark.org`.

[103] Gokul Soundararajan et al. “Dynamic Resource Allocation for Database Servers Running on Virtual Storage”. In: *FAST*. 2009.

[104] *Tuning Java Garbage Collection for Apache Spark Applications*. `https://goo.gl/SIWlqx`.

[105] Ion Stoica, Scott Shenker, and Hui Zhang. “Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks”. In: *SIGCOMM’98*.

[106] Ion Stoica et al. “A proportional share resource allocation algorithm for real-time, time-shared systems”. In: *RTSS’96*.

[107] *TPC-H*. `http://www.tpc.org/tpch`.

[108] *Tuning Spark*. `https://spark.apache.org/docs/latest/tuning.html\#garbage-collection-tuning`.

[109] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet another resource negotiator”. In: *SoCC*. 2013.

[110] Shivaram Venkataraman et al. “Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics”. In: *NSDI*. 2016.

[111] Abhishek Verma et al. “Large scale cluster management at Google with Borg”. In: Eurosys’15.

[112] Carl A Waldspurger and William E Weihl. “Lottery scheduling: Flexible proportional-share resource management”. In: *OSDI’94*.

[113] Carl A Waldspurger and William E Weihl. “Stride scheduling: deterministic proportional-share resource management”. In: *MIT Tech Report, 1995*.

[114] Qian Wang et al. *NADSort*. `http://sortbenchmark.org/NADSort2016.pdf`.

[115] *X1 instances*. `https://aws.amazon.com/ec2/instance-types/x1/`.

[116] M. Zaharia et al. “Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters”. In: *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association. 2012.

[117] M. Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Proc. NSDI*. 2011.

[118] Matei Zaharia et al. “Improving MapReduce Performance in Heterogeneous Environments”. In: *OSDI*. 2008.

[119] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *NSDI’12*.