Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications

Sangmin Lee Zhenhua Guo Omer Sunercan Jun Ying Thawan Kooburat
Suryadeep Biswal Jun Chen Kun Huang Yatpang Cheung Yiding Zhou
Kaushik Veeraraghavan Biren Damani Pol Mauri Ruiz Vikas Mehta Chunqiang Tang

Facebook Inc.

Abstract

Sharding is widely used to scale an application. Despite a decade of effort to build generic sharding frameworks that can be reused across different applications, the extent of their success remains unclear. We attempt to answer a fundamental question: *what barriers prevent a sharding framework from getting adopted by the majority of sharded applications?*

We analyze hundreds of sharded applications at Facebook and identify two major barriers: 1) lack of support for geo-distributed applications, which account for most of Facebook's applications, and 2) inability to maintain application availability during planned events such as software upgrades, which happen ≈1000 times more frequently than unplanned failures. A sharding framework that does not help applications to address these fundamental challenges is not sufficiently attractive for most applications to adopt it. Other adoption barriers include the burden of supporting many complex applications in a one-size-fit-all sharding framework and the difficulty in supporting sophisticated shard-placement requirements. Theoretically, a constraint solver can handle complex placement requirements, but in practice it is not scalable enough to perform near-realtime shard placement at a global scale.

We have overcome these adoption barriers in Facebook's sharding framework called *Shard Manager*. Currently, Shard Manager is used by hundreds of applications running on over one million machines, which account for about 54% of all sharded applications at Facebook.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Software and its engineering → Distributed systems organizing principles**.

Keywords: shard management, sharding, availability

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SOSP '21, October 26–29, 2021, Virtual Event, Germany

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483546

1 Introduction

Sharding is a common strategy to scale an application by dividing its work into smaller units and assigning them to different servers. For example, consider a key-value store with ten billion key-value pairs. We can divide the store into 100K shards, each containing 100K key-value pairs, and then distribute the 100K shards across 1,000 servers. Sharding is ubiquitous—GitHub currently hosts over 5,000 repositories that mention "shard" and Facebook internally has hundreds of sharded applications.

A proper sharding implementation needs to solve many hard problems such as shard placement, load balancing, shard discovery & request routing, and application lifecycle management. To avoid duplicate efforts, there has been a decade of effort to build generic sharding frameworks [2, 4, 12, 25, 27, 35, 54] that can be reused across different applications, but the extent of their success remains unclear.

Proprietary sharding frameworks often propose a new programming model and a selective set of features. However, without knowing their adoption trajectory and market share (i.e., 1% or 90%), it is hard to evaluate whether their programming model and feature set are flexible or rich enough to support the majority of sharded applications. In particular, the best-known sharding frameworks, Google's Slicer [4] and Microsoft's Azure Service Fabric (ASF) [35], differ significantly in their programming model, feature set, design, and implementation. Their divergence and the difficulty in comparing their success using common metrics such as the adoption rate make it hard for others to choose the best practices to follow.

Open-source sharding frameworks have a limited adoption rate. Twitter's Gizzard [25] and Uber's Ringpop [54] were already archived. LinkedIn's Helix [27] is better known; nonetheless GitHub hosts only two tryout Helix applications [23, 44]. None of the well-known open-source projects including Cassandra, HBase, Redis, MongoDB, Couchbase, and Kafka, share a common sharding framework.

1.1 Adoption Barriers for Sharding Frameworks

What barriers prevent a sharding framework from getting adopted by the majority of sharded applications? We share takeaways from our analysis of all sharded applications at Facebook, which amount to hundreds.

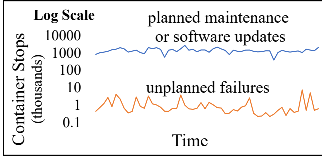

Figure 1. Planned vs. unplanned container stops.

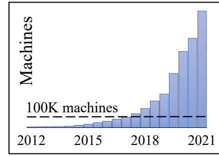

Figure 2. Machines used by SM applications.

First, improving availability is a major motivator for applications to adopt a sharding framework, but this requirement is not well supported by existing sharding frameworks. They all support shard failover, but they use it to handle both unplanned failures (e.g., power loss or process crash) as well as planned events (e.g., hardware maintenance or software upgrades). Figure 1 shows that at Facebook, container stops due to planned events are ≈1000 times more frequent than unplanned failures. Treating planned events as failures amplifies unavailability by ≈1000x as every failover causes a period of shard unavailability. Moreover, existing sharding frameworks cannot perform shard migration (e.g., due to load balancing) without impacting in-flight client requests, which further impairs availability. As a result, existing sharding frameworks often bear the cost of deploying extra shard replicas to compensate for their lower per-replica availability.

Second, existing sharding frameworks do not provide sufficient support for geo-distributed applications, which are ubiquitous in large-scale internet services. Existing sharding frameworks are incapable of globally coordinating operations across multiple regional cluster managers, e.g., to prevent two independent container restarts in two geographic regions from accidentally bringing down two replicas of the same shard. Moreover, they cannot spread a shard's replicas across regions for better resilience, cannot migrate shards across regions for better load balancing, and disallow individual shards from specifying regional placement preferences for better network locality. As a result, they often bear the cost of statically deploying extra shard replicas to multiple regions in order to compensate for their lack of support for geo-distributed applications.

Lack of support for geo-distributed applications and inability to maintain application availability during planned events, fundamentally limit the adoption of existing sharding frameworks. Our analysis in §2.3 shows that 1) 100% of Facebook's sharded applications are deployed to multiple regions and 2) availability improvements play an even bigger role than shard placement & load-balancing (PLB) features in motivating the adoption of a sharding framework. Existing sharding frameworks often emphasize PLB, but barely touch on these two most important aspects. Without gaining significant benefits from a sharding framework on these two most important aspects, application owners might not be motivated to adopt the sharding framework due to the downsides of taking on a deep and complex dependency that is directly linked into their application, which can present reliability risks and make it difficult to troubleshoot production issues.

Compared with the two barriers above, the two barriers described below are less fundamental but further lower the adoption rate of sharding frameworks.

Third, many applications require advanced PLB features—unfortunately, most sharding frameworks use hand-crafted PLB heuristics, which are easy to start with but become brittle and hard to extend over time. As a result, it is hard to add new PLB features required by new applications (e.g., region-aware shard placement required by geo-distributed applications), which further hinders adoption. By contrast, it is much easier to introduce a new PLB feature by adding new constraints to an optimization problem and then solving it with a generic constraint solver [24]. Unfortunately, a solver on its own is not sufficiently scalable to handle near-realtime PLB, especially for large-scale geo-distributed applications.

Fourth, existing sharding frameworks do not account for the bimodal nature of their workloads, i.e., many small applications plus a few mega applications. Small applications prefer simplicity, whereas mega applications often employ highly customized features. Attempting to implement all custom features in a single sharding framework makes it unwieldy and overly complex.

1.2 Our Contributions

In this paper, we present Facebook's shard-management framework called *Shard Manager (SM)*. Since its production deployment in 2012, the server side of SM applications alone (i.e., excluding the client side) has grown to consume over one million machines (Figure 2). Currently, SM applications amount to hundreds and account for about 54% of Facebook's sharded applications, which likely far exceeds other sharding frameworks' adoption rate. SM applications process billions of requests per second, which is ≈100 times higher than the reported request rate of Slicer applications [4, 49]. Examples of SM applications include a Paxos-based database [45], a blob store [8], a queue service [47], a data bus [36], a machine learning (ML) inference platform [50], a ML-training control plane [21], ML feature stores, a time series database [51], a stream processing engine [46], a pub/sub system [56], etc.

This paper makes the following contributions:

- We analyze all sharded applications at Facebook and present our findings to guide the design of future sharding frameworks.
- We propose a simple yet powerful programming model and share our internal adoption data to prove its capability in supporting the majority of sharded applications.
- We identify two barriers that fundamentally limit the adoption of existing sharding frameworks: 1) inability to maintain application availability during planned events and 2) lack of support for geo-distributed applications. To maintain availability, SM negotiates with the underlying

cluster managers about when to safely execute container lifecycle operations. Moreover, SM ensures that an application drops no requests during a graceful shard migration. To support geo-distributed applications, SM handles global shard placement & migration and does global coordination across multiple regional cluster managers.

- SM uses a constraint solver for near-realtime shard placement at a global scale. We share the optimizations that enable our solver to scale. The expressiveness of a constraint solver allows us to easily add new PLB features for new applications, which further boosts SM's adoption.
- We propose a composable SM ecosystem that allows certain complex applications to bring their own components. For example, while most applications adopt the whole SM framework without customization, a complex SQL database may choose to adopt SM's PLB component but implement its own application lifecycle manager. Note that all SM adoption numbers in this paper exclude applications that have adopted some SM components but not the whole SM framework.

The rest of the paper is organized as follows. §2 reports the demographics of real-world sharded applications. §3 provides an overview of SM. §4 presents how SM maintains application availability. §5 describes SM's allocator. §6 presents how we make SM scalable and fault tolerant. §7 discusses our ongoing work to further boost SM's adoption. §8 evaluates SM. §9 discusses related work. §10 concludes the paper.

2 Analysis of Sharded Applications

We spent months of efforts on analyzing all sharded applications at Facebook. This is a daunting task due to Facebook's huge code base and thousands of applications. We did code search and leveraged sharded applications' common traits to identify a candidate pool of sharded applications as exhaustively as possible. We then read through their code and interviewed dozens of application owners as needed to filter out non-sharded ones. Below, we introduce sharding basics (§2.1), summarize the demographics (§2.2) of all sharded applications at Facebook, which amount to hundreds, explain why certain applications might not adopt certain sharding frameworks (§2.3), discuss data persistency (§2.4), and finally describe some real examples of sharded applications (§2.5).

2.1 Sharding Basics

Sharding divides a key space (e.g., strings or 128-bit integers) into shards of non-overlapping key ranges and assigns each shard to an application server that handles all client requests targeting the keys in the shard. An application server may host multiple shards. In response to load changes, a shard may migrate from one application server to another in order to balance load across servers. As the shard-to-server assignment changes, the shard map needs to be disseminated to all

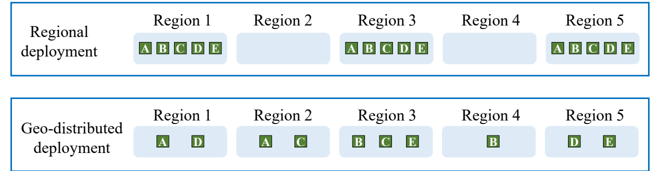

Figure 3. Regional vs. geo-distributed deployment. An application with 5 shards (A-E) is deployed to 5 regions (1-5).

application clients in a timely manner so that the clients can send requests to the correct application servers.

A shard may have multiple geo-distributed replicas in order to provide resilience against large-scale failures, to shorten network latency to certain clients, or simply to increase the request-handling throughput of the shard. The placement of shard replicas on application servers needs to consider all these factors in addition to balancing load.

To maintain shard availability, applications need to carefully prepare for and handle both unplanned failures (e.g., power loss) and planned events (e.g., software upgrades). The latter is particularly important because it is ≈1000x more frequent than the former (Figure 1). For example, restarting an application server that hosts the primary replica of a shard without first draining the primary replica out of the server, might cause the shard to be unavailable during the restart.

2.2 Demographics of Sharded Applications

2.2.1 Sharding Schemes

In Figure 4, we show the breakdown of all sharded applications at Facebook. 54% of them are built atop SM; we discuss how to further boost SM's adoption in §7. *Custom sharding* represents a small number of largest and most complex data stores that have their own sharding control plane, including a SQL database [52], a graph data store [10], and a log store [43]. These large applications account for only 1% of sharded applications but 27% of server usage.

Static sharding uses a fixed binding between shards and containers. Facebook's cluster manager Twine [60] deploys an application as a group of containers called *tasks*. The *taskIDs* are indexed sequentially from zero and are often used for static sharding. For example, the task with $taskID = key \mod total_tasks$ is responsible for the *key*. Despite the theoretical advantage of consistent hashing, static sharding is ≈3x more popular than consistent hashing, indicating that resharding is rare and the overhead of resharding is not prohibitive. For example, it is common that after a resharding, an application can rebuild its soft state from an external persistent store.

Next, we report various properties of applications built atop SM to understand why they use a sharding framework.

2.2.2 Regional vs. Geo-distributed Deployment

Facebook operates out of tens of geo-distributed *regions*. Each region consists of multiple *data centers*. All applications

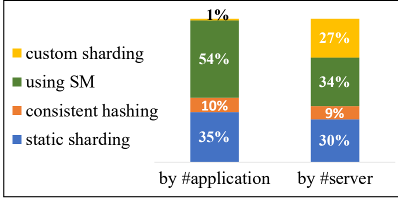

Figure 4. Breakdown of all sharded applications at Facebook.

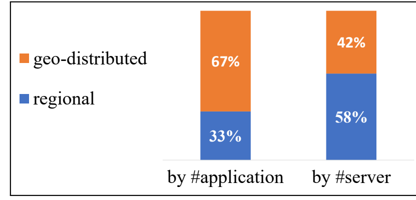

Figure 5. SM applications' usage of regional & geo-distributed deployments.

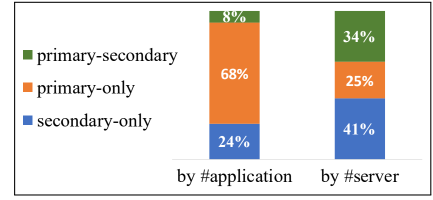

Figure 6. SM applications' usage of different shard replication strategies.

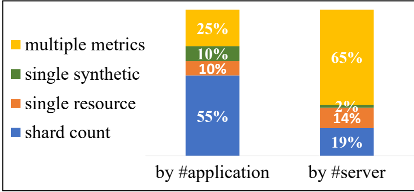

Figure 7. SM applications' usage of load balancing policies.

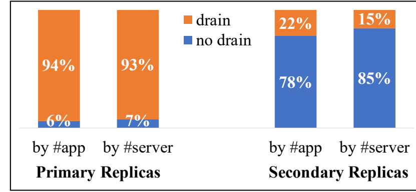

Figure 8. SM applications' usage of drain policies for container restarts.

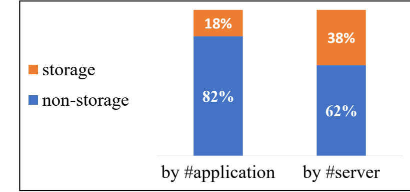

Figure 9. SM applications' usage of storage and non-storage machines.

are deployed to multiple regions in order to provide resilience against whole-region outages and to reduce network latency to global users. An application can be deployed in different modes (Figure 3). With a *regional deployment*, a complete copy of all shards of the application are hosted in one region. A regional deployment may be duplicated in multiple regions to provide redundancy, but its shards cannot migrate from one region to another. With a *geo-distributed deployment*, a complete copy of all shards might not exist in any single region, and its shards can migrate across regions on demand.

A geo-distributed deployment may use fewer replicas per shard than the equivalent regional deployments, thereby saving hardware capacity. In Figure 3, suppose *region 1* fails. With a geo-distributed deployment, the shards originally in *region 1* can be redistributed across multiple regions that have any amount of available capacity. With a regional deployment, it is harder to find a single region that has enough capacity to accommodate a complete copy of all shards. As a result, application owners often over-provision duplicate copies of regional deployments ahead of time.

Figure 5 compares the adoption rate of regional vs. geo-distributed deployments. Regional deployments are still popular even if global deployments offer more advantages. Historically, many applications started with duplicate regional deployments in every region. This approach is simple, but wastes capacity and is unsustainable as the number of regions grows rapidly. In recent years, we have observed a strong trend of applications migrating from regional deployments to geo-distributed deployments.

2.2.3 Shard Replication Strategies

We first define some terminology related to the leader-follower programming paradigm. A shard can have multiple replicas and the *role* of a replica is either *primary* or *secondary*. A shard can have at most one primary replica plus an arbitrary number of secondary replicas. We classify sharded applications into three categories.

- *Primary-only* (no secondary): Each shard has a single replica. SM guarantees that no two servers serve the same shard at the same time.
- *Secondary-only* (no primary): Each shard has multiple replicas that all play an equal role.
- *Primary-secondary*: Each shard has one SM-elected primary replica plus one or more secondary replicas. The primary often handles writes.

Figure 6 shows the usage of primary and secondary roles. Primary-only applications are popular for the following reason. Many traditional applications have one active leader plus multiple hot/cold standbys. Applications using hot standbys map to SM's primary-secondary applications, whereas applications using cold standbys map to SM's primary-only applications. A primary-only SM application does not need to keep cold standbys because upon the failure of a shard's primary, SM can immediately recreate the shard's new primary in another running container that currently hosts other shards' primaries. In other words, the unused capacity of the application's running containers serves as cold standbys. The popularity of primary-only applications implies that cold standbys are more popular due to their simplicity.

2.2.4 Load-Balancing Policies

SM periodically collects load information of shards from application servers and balances load by moving shards out of overloaded servers (§5). Figure 7 shows the usage of different load-balancing (LB) policies. "*Shard count*" means doing LB based on the number of shards per server. "*Single resource*" means doing LB based on a single resource metric such as

CPU, memory, or storage. "*Single synthetic*" means doing LB based on an application-level metric such as request queue size. "*Multiple metrics*" means doing complex LB based on multiple resource/synthetic metrics. 20% of applications use single synthetic/resource LB because they have a single dominating bottleneck. 55% of applications use shard-count-based LB because their shard load is sufficiently uniform. Applications using multi-metric LB account for 65% of server usage, indicating that sharding frameworks need to support advanced LB features. Moreover, out of all servers used by geo-distributed deployments (Figure 3), 33% of them are for applications that dictate regional shard-placement preferences, thus requiring region-aware placement features.

2.2.5 Drain Policies

When a container needs to restart in place on a machine, e.g., to upgrade an application's executable, SM can either proactively drain the application's shards out of the impacted container or do nothing, i.e., tolerating the downtime of the shards. Figure 8 shows that most applications choose to drain their primaries but not their secondaries. A shard has at most one primary, which often plays an important role and requires high availability. By contrast, a shard can have multiple secondaries and SM can manage the pace of container restarts to ensure that a minimum number of secondary replicas per shard is always available.

2.2.6 Usage of Storage Machines

Figure 9 compares the usage of storage (i.e., SSD/HDD) and non-storage machines. SM applications consume a higher percentage (38%) of storage machines than average applications do because many sharded applications have states and need storage (§2.4). Applications using storage machines need extra support. For them, it is especially important to spread a shard's replicas across large fault domains to ensure data availability. Moreover, as they often take longer to recover from a hard crash, it is important to handle their planned maintenance events gracefully.

2.3 Using Data to Elaborate on Adoption Barriers

In §1.1, we argue that lack of support for geo-distributed applications and inability to uphold application availability during planned events fundamentally limit the adoption of existing sharding frameworks. We elaborate on them below.

If a sharding framework does not uphold application availability during planned events, the majority of sharded applications might not adopt it. Specifically, combining data in Figures 6 and 8, we see that about 70% of SM applications choose to gracefully drain shards before a container restart. These applications might not adopt a sharding framework that cannot gracefully handle planned events. Moreover, 55% of SM applications that use shard-count-based LB (Figure 7) might not adopt such a framework either since static sharding is sufficient for their LB needs and they do not need the sharding framework's basic failover function. The cluster manager already provides container-level failover—if a machine or container is not responsive, the cluster manager restarts the container on another machine. These applications using shard-count-based-LB adopted SM primarily because of its advanced capability in upholding application availability during planned events.

If a sharding framework does not provide sufficient geo-distribution support, the majority of sharded applications might not adopt it. Specifically, 67% of SM applications that use geo-distributed deployments (Figure 5) might not adopt it. Moreover, each of the remaining 33% is set up as multiple regional deployments, which may need global coordination. Suppose an application uses two regional deployments, each shard has one replica per region, and the two regions' local cluster managers independently restart two containers at the same time. Those two containers might happen to host a shard's different replicas, causing the shard's both replicas to be unavailable. Some SM regional deployments work around this problem by having their owners perform operations sequentially in one region at a time. It is still problematic because 1) the number of regions grows quickly, prolonging the operation duration, and 2) some infrastructure-level maintenance operations kick off automatically without going through application owners. As a result, we see a strong trend of migration from regional to geo-distributed deployments.

2.4 Data-Persistency Options

Sharded applications often need to store and access data. ASF's strongly-consistent Reliable Collections [35] help build local-storage-based persistent stores directly into applications. Slicer's follow-up work [3] advocates that a sharding framework should link a custom library into an application to manage its data persistency and replication. Our experience, however, suggests otherwise.

Our colleagues initially developed a Paxos [38] library, hoping it would be used along with SM to build many applications. However, it eventually had only one use case, i.e., ZippyDB [45] described in §2.5. In practice, most applications do not need strong consistency. Applications that do need strong consistency almost always prefer the simple solution of a primary replica accessing external databases.

Even if eventually-consistent applications do not need Paxos, they still need to handle data updates across replicas. Our colleagues initially developed a library for asynchronous data transfer among the replicas of a shard, but it did not get a wide adoption. In practice, different replicas of a shard often directly obtain data updates from off-the-shelf external tools such as a Kafka-like data bus or an HDFS-like file system.

Our experience suggests that in practice application developers heavily prioritize simplicity. Whenever performance and costs permit, we recommend applications to use a data-persistency method in the list below, ranked in order of increasing complexity and hence decreasing preference.

1. **Stateless:** An application directly operates on external

databases. Most stateless applications are not sharded, but some are sharded in order to solve the high-fanout problem, i.e., every application server talking to every database shard, which hinders database connection reuses in a large application.

2. **Soft state:** An application caches external stores' persistent states in memory for fast access. Soft-state applications often rely on sharding to ensure that operations related to a specific data item in the external store always go through the same application server.
3. **Standard materialized state:** An application stores on a local SSD materialized-view-style state derived from external persistent stores. It obtains data updates via *standard external tools* such as a Kafka-like data bus. In case of a total data loss, application states on the local SSD can be rebuilt from the external persistent stores.
4. **Custom materialized state:** It works the same as above except that the application uses a built-in *custom library* to obtain data updates and keep its replicas eventually consistent. This approach is recommended by Slicer's follow-up work [3], but it is rarely used at Facebook even though we have provided such a custom library.
5. **Persistent state:** An application manages its own replicated persistent states on local SSDs via a consensus protocol. This approach is akin to ASF's strongly-consistent Reliable Collections [35], but few applications at Facebook use it even though we have provided a Paxos library.

Applications using option 1 or 2 do not need storage machines. They are the majority; as shown in Figure 9, they account for 82% of sharded applications and 62% of machine usage. Option 4 is rarely used. We argue that, for simplicity, most applications that want to use option 4 can and should use option 3 instead. Option 5 is only used to build very few persistent stores such as ZippyDB [45], which then support many other applications that use option 1, 2, 3, or 4. Option 5 needs to handle complex issues such as continuous data-consistency auditing to guard against bit rot. We argue that most applications that want to use option 5 can and should use option 1 or 2 instead. For example, among cluster managers, Borg [64] implements its own Paxos-based embedded persistent store (option 5), whereas Kubernetes [37], Twine [60], and Protean [32] all successfully delegate data persistency to external databases (option 2).

2.5 Put It Together via Example Applications

We describe two applications to put all the concepts together. *AdEvents* are a group of stream-processing applications directly related to revenue generation. They use option 3 in §2.4 and obtain updates via a Kafka-like data bus. Initially, they were statically sharded, used regional deployments, and needed standby deployments in multiple regions to guard against whole-region outages. The standby deployments often remained underutilized. They were converted to primary-only SM applications, using geo-distributed deployments. Thanks to better load balancing, flexible shard placement, and dynamic shard migration across regions, SM helped reduce their machine usage by 67%.

ZippyDB [45] is a Paxos-based geo-distributed database, using data-persistency option 5 in §2.4. It was started on SM in 2013 and has grown to become one of the most widely used databases at Facebook. Each ZippyDB shard has a primary serving as the Paxos leader and proposer, and multiple secondaries serving as acceptors and learners. Shard replicas can be placed at different regions for high availability. Load balancing is based on multiple metrics, including CPU, storage, and shard count. To reduce hardware costs, most ZippyDB deployments use one primary plus only two secondaries per shard and rely on SM to carefully handle maintenance operations to avoid losing two replicas at the same time (§4).

3 SM Overview

This section provides an overview of SM's sharding abstraction, architecture, programming model, and major features.

3.1 Sharding Abstraction

Sharding divides a key space into shards. A sharding framework needs to make two fundamental decisions:

- Should it directly shard an application-provided key space (so-called *app-key* approach), or hash application keys into universally unique identifiers (UUID) and then shard the UUID key space (so-called *UUID-key* approach)?
- Who decides how to shard the key space, the application (so-called *app-sharding* approach) or the sharding framework (so-called *framework-sharding* approach)? The latter allows the framework to freely split or merge shards during placement & load balancing (PLB) operations.

Slicer chooses *UUID-key* and *framework-sharding*, whereas ASF and SM choose *app-key* and *app-sharding*. The trade-off is that ASF and SM's approach supports a broader set of applications but makes PLB harder for the framework. For example, an SM application may choose to use three uneven shards denoted by shardID:[key_range] as follows: S0:[1,9], S1:[10,99], S2:[100,100000]. We describe how SM addresses the PLB challenges in §5.

Slicer's *UUID-key* approach destroys key locality. On one hand, it helps spread adjacent hot application keys evenly across the UUID key space. On the other hand, without key locality it is impossible to support certain popular operations such as prefix scans in a key-value store. For example, Facebook's eventually-consistent key-value store *Laser* [15] is built atop SM and processes nearly one billion queries per second at peak; 9% of those queries are prefix scans.

Slicer's *framework-sharding* approach makes it hard to support certain applications. For example, Laser runs a daily

MapReduce job to partition data into shards and build per-shard indices. The data and indices are daily reloaded into Laser for serving. If SM dynamically splits or merges shards, they would be misaligned with the indices produced by MapReduce. By contrast, SM's *app-sharding* approach allows an application to decide the key-to-shard mapping and set different policies for each shard, e.g., per-shard regional placement preference. Moreover, SM's *app-sharding* approach allows applications to use other systems [5, 6, 55] to intelligently group keys into shards based on various constraints such as data-access pattern and region capacity, and then instruct SM to place shards accordingly.

3.2 SM Architecture

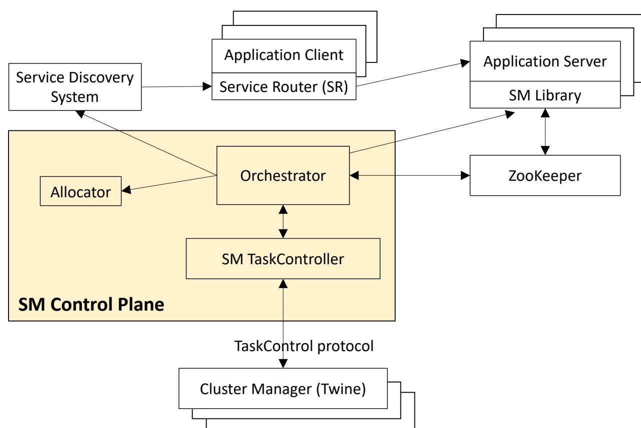

Figure 10. Simplified diagram of the SM ecosystem.

Figure 10 depicts the SM ecosystem. An application server, written by an application owner and running in a container, hosts one or more shards assigned by the orchestrator. The *SM library* is linked into application servers and takes commands from the orchestrator. The *service router* library is linked into application clients. It learns from the service discovery system about which application server is responsible for which shards and routes requests accordingly.

The *orchestrator* monitors the health and resource consumption of shards assigned to application servers. When an application server fails or its load changes, the orchestrator invokes the *allocator* to generate a new shard-to-server assignment. It distributes the new shard map to application clients via the service discovery system, which internally uses a multi-level data-distribution tree to fan out. The orchestrator makes direct RPC calls to application servers for load-information collection and shard-assignment notification via APIs described in §3.3.

Twine, Facebook's cluster manager, informs SM's *TaskController* of upcoming hardware maintenance events, kernel updates, and container starts/stops/moves via the TaskControl protocol [60]. If needed, the TaskController works with the orchestrator to drain shards out of the affected server before allowing a container operation to proceed.

ZooKeeper serves multiple purposes. First, it stores the orchestrator's persistent state. Second, during start-up, an application server reads its shard assignment from ZooKeeper, without dependency on the SM control plane. Third, the orchestrator watches the SM-library-created ephemeral nodes in ZooKeeper to detect failures of application servers.

3.3 SM Programming Model

```
add_shard(shardID, role /* primary or secondary */)
drop_shard(shardID)
change_role(shardID, current_role, new_role)
prepare_add_shard(shardID, current_owner, role)
prepare_drop_shard(shardID, new_owner, role)
```

Figure 11. APIs implemented by an application server and invoked by the orchestrator. SM supports all three primary/secondary replication strategies described in §2.2.3.

SM presents a very simple programming model to lower the adoption barrier. An application implements the APIs in Figure 11. *add_shard()* and *drop_shard()* are implemented by all SM applications. *change_role()* is implemented by those using both primary and secondary shard-replica roles. *prepare_add_shard()* and *prepare_drop_shard()* are described in §4.3. Application clients use the following APIs to call the application servers:

```
rpc_client = get_client(app_name, key)
rpc_client.function_foo(...)
```

3.4 SM Features

Below is a list of SM's main features.

- **Supporting geo-distributed applications**. SM supports both regional and geo-distributed deployments (§2.2.2).
- **Application lifecycle management** (§4). SM negotiates with the cluster managers in one or more regions about when to safely execute container lifecycle operations.
- **Graceful shard migration** (§4.3). SM can do live primary-replica migration without dropping any request in flight.
- **Regional placement preference** (§5.1). SM allows fine-grained control of placing an application's different shards at different regions for better locality.
- **Placement spread** (§5.1). For better availability, SM can spread a shard's replicas across fault domains at all levels, including regions, data centers, and racks.
- **Load balancing** (§5). SM can use multiple metrics to balance load across heterogeneous application servers that have different capacities.
- **Shard scaling**. In response to load changes on shards, SM can adjust each shard's replica count independently.

- **Automatic failover**. Upon the failure of an application server, SM reassigns its shards to other servers.

4 Maximize Application Availability

SM provides two major benefits to applications: availability improvements and intelligent shard placement. These features are also the foundation for supporting geo-distributed applications. This section focuses on availability improvements. SM improves availability in multiple ways:

1. SM places a shard's replicas across fault domains (§5).
2. SM's TaskController collaborates with the underlying cluster managers to selectively delay negotiable container lifecycle operations that are unsafe (§4.1).
3. SM proactively prepares for non-negotiable events such as hardware maintenance and kernel upgrades (§4.2).
4. SM ensures that no requests are dropped from an application during a graceful primary-replica migration (§4.3).

4.1 Graceful Handling of Negotiable Events

SM never approves unsafe operations when handling negotiable events such as application upgrades or an auto-scaler adjusting an application's container count in response to load changes.

Periodically, Twine [60], Facebook's cluster manager, notifies SM's TaskController of a set of pending container operations (start/stop/restart/move) and SM's TaskController responds with a subset of approved operations that will not endanger the availability of any shard. Twine delays the execution of unapproved operations, but executes the approved operations immediately. When those operations finish, Twine notifies SM's TaskController so that it can approve the next batch of operations.

SM's TaskController enforces an application's preconfigured policy that specifies 1) in the event of a container operation, whether to drain shards out of the impacted container or leave them there, depending on the operation's impact and duration, 2) a global cap on the number of allowed concurrent container operations, and 3) a per-shard cap on the number of replicas that are allowed to be temporarily unavailable. These two caps account for the containers and shard replicas that are already unavailable due to ongoing unplanned outage such as hardware failures. Guided by SM's knowledge of the shard-to-container assignment, the TaskController carefully calculates a maximum set of container operations that do not violate either the global cap or the per-shard cap and notifies Twine to execute them.

For a geo-distributed application (§2.2.2) jointly hosted by multiple Twine instances, SM's TaskController receives notifications from and enforces the caps across all involved Twine instances. Suppose the application's per-shard cap is one, and two Twine instances independently plan to restart two containers in different regions, which happen to host two replicas of the same shard. SM's TaskController can approve one Twine instance to proceed with its container restart while informing the other Twine instance to wait, to avoid losing both the shard's replicas at the same time.

4.2 Graceful Handling of Non-negotiable Events

SM cannot delay hardware maintenance or kernel upgrades; otherwise, fleet-wide maintenance cannot finish in time. Twine gives SM an advanced notice of the start and end time of a maintenance event and the impact of the event, which could be network unavailability, runtime state loss, full state loss, and full machine loss. How SM handles an event depends on its impact and the application's configuration. For example, if a rack switch maintenance only causes network loss for a short period of time, SM may allow secondary replicas to stay on the affected machines and demote the primary replicas on those machines while promoting their corresponding secondary replicas on unaffected machines to become primaries.

4.3 Graceful Primary-Replica Migration

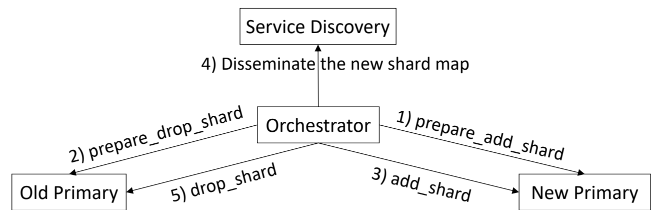

Figure 12. Graceful primary-replica migration.

Since a shard's primary replica often carries important responsibilities such as handling writes, SM takes extra caution to ensure zero downtime during a graceful primary migration. It uses the APIs in Figure 11 to instruct the old primary P_{old} to forward client requests to the new primary P_{new} until the migration finishes, as depicted in Figure 12.

1. Through *prepare_add_shard()*, SM informs P_{new} to prepare for taking over the primary role. P_{new} processes a primary-related request (e.g., write) only if the request is forwarded from P_{old}. Application clients still send requests to P_{old}.
2. Through *prepare_drop_shard()*, SM informs P_{old} that P_{new} will take over the primary role. Then P_{old} starts to forward all primary-related requests to P_{new}.
3. Through *add_shard()*, SM informs P_{new} that it now officially holds the primary role and can accept primary-related requests directly from application clients.
4. SM instructs the service discovery system to notify application clients to send future requests to P_{new}.
5. Through *drop_shard()*, SM informs P_{old} that its replica is no longer needed. P_{old} keeps forwarding client requests to P_{new} and drops its replica when no more requests arrive.

Throughout the migration process, no client request is dropped.

The SM orchestrator makes direct RPC calls to application servers to precisely control the operation sequence above. By contrast, Slicer's application servers do not directly communicate with Slicer's controller [4]. Hence, Slicer cannot orchestrate live migration that requires precise ordering of operations performed by distributed application servers.

5 Shard Placement and Load Balancing

After presenting availability improvements in §4, this section presents the other major benefit that SM offers to its applications, i.e., intelligent shard placement. This capability is an important foundation for supporting geo-distributed applications. Below, we describe the shard-placement constraints (§5.1), how to translate the constraints into an optimization problem (§5.2), and how to scale SM's constraint solver (§5.3).

5.1 Hard Constraints and Soft Goals

We design the SM allocator to optimize application's availability, performance, and efficient use of hardware, in that order. The allocator honors hard placement *constraints* and a prioritized list of soft optimization *goals*.

Below are some of the hard constraints.

1. **System stability**: Cap the number of concurrent shard moves per server and per application to limit churns that may threaten system stability. Similarly, cap the number of a shard's replicas that can be moved concurrently.
2. **Server capacity**: For each load-balancing (LB) metric (e.g., CPU), the aggregate consumption of all shards on a server should not exceed the server's capacity.

Below are some of the soft goals presented from high to low priority.

1. **Region preference**: SM allows a fine-grained control of placing individual shards at different regions.
2. **Spread of replicas**: For better availability, it is preferred to place a shard's replicas across fault domains at all levels, including regions, data centers, and racks.
3. **Planned maintenance**: Depending on an application's configuration, SM may proactively drain shards out of a server with pending maintenance or software upgrades.
4. **Utilization threshold**: An application may prefer its server utilization to not exceed a threshold, e.g., 90%.
5. **Global load balancing**: Balance load across servers in different regions. For example, no server's utilization should exceed the average utilization of all servers by 10%.
6. **Regional load balancing**: Balance load across servers within a region.
7. **Parallel shard failover**: Evenly distribute shards on a failed server to multiple other servers for faster recovery.

Shard allocations are performed in either an *emergency mode* or an *periodic mode*. The emergency mode is triggered upon detecting unavailable shards, e.g., due to server failures. It tries to place unavailable shards as quickly as possible while satisfying hard constraints, but may temporarily deteriorate soft goals. The periodic mode runs regularly, takes a longer time to optimize the placement of all shards, and must not deteriorate soft goals. The separation of the two modes helps SM to perform near-realtime allocations at scale.

5.2 Using a Generic Constraint Solver

SM's allocator initially used hand-crafted placement heuristics for years. Over time, the heuristics became complex, brittle, and hard to extend. We then started a project to rewrite it with yet another supposedly simpler heuristic implementation. Halfway through, we abandoned the project because the heuristics again became overly complicated as we started to add the rich features described in §5.1. Finally, we decided to switch from heuristics to a generic constraint solver.

SM's placement constraints and goals can be formulated as a constrained optimization problem in a mathematical form. For ease of use, we adopt a popular constraint solver at Facebook called *ReBalancer*, which provides a high-level API interface to specify constraints and goals. In Figure 13, statements 1 & 2 specify capacity constraints on host-level CPU and rack-level network, respectively. Statements 3 & 4 specify soft LB goals on CPU and network; CPU is considered more important because its weight of 1.0 is higher than network's weight of 0.5. Statements 5 & 6 specify a goal of placing *shard1* in *regionA* and a stronger (weight 2.0) goal of placing *shard2* in *regionB*. Statements 7 & 8 specify a goal of spreading *shard3*'s replicas across regions. Placement features like those in statements 5–8 help support geo-distributed applications. Internally, ReBalancer translates these high-level statements into a constrained optimization problem's mathematical form.

Behind its uniform API interface, ReBalancer has the freedom of choosing different backend solvers to solve different optimization problems. For example, at Facebook, RAS [48] uses ReBalancer with a mixed-integer programming (MIP) solver to allocate data-center resources. However, MIP is not

```
1: addConstraint(CapacitySpec{.scope="host", .metric="cpu"});
2: addConstraint(CapacitySpec{.scope="rack", .metric="network"});
3: addGoal(BalanceSpec{.scope="host", .metric="cpu''}, 1.0);
4: addGoal(BalanceSpec{.scope="rack", .metric="network''}, 0.5);
5: shardAffinity = {
       {"shard1", "regionA", 1.0},
       {"shard2", "regionB", 2.0},
   }
6: addGoal(AffinitySpec.scope="region", .affinities=shardAffinity)
7: replicaMap = {
       {"shard3_replica1", "shard3"},
       {"shard3_replica2", "shard3"},
   }
8: addGoal(ExclusionSpec.scope="region", .partition=replicaMap);
```

Figure 13. Examples of specifying hard placement constraints and soft goals using the ReBalancer APIs.

sufficiently scalable for SM's use case. Specifically, MIP can solve an optimization problem with millions of assignment variables in tens of minutes, whereas SM needs to solve a problem with billions of variables in tens of seconds.

5.3 Scaling the Constraint Solver

We scale the constraint solver via multiple techniques:

1. SM divides a large application into *partitions* (§6.1), builds a smaller optimization problem for each partition separately, and solves them on multiple machines in parallel.
2. We configure ReBalancer to use *Local Search* [1] instead of MIP to solve each partition's optimization problem while meeting our rigid time constraint.
3. ReBalancer has accumulated many domain-independent improvements to accelerate local search.
4. SM's allocator provides domain-specific knowledge to guide ReBalancer to further accelerate local search.

We elaborate on the last three techniques below.

First, we use local search to speed up the solver. Starting from the current shard assignment, it considers moving shards from hot servers to cold servers by prioritizing shards whose constraint or goal violations impair the optimization objective the most. It evaluates a large number of such shard moves and keeps the best one. Local search repeats until it either cannot find improvements or uses up a predetermined time and move budget.

Second, as a widely used solver at Facebook, ReBalancer has accumulated many improvements over the years. Specifically, to reduce the number of variables in an optimization problem, it figures out from the mathematical formula which shards are equivalent to one another and reuses the computation for equivalent shards. Moreover, it represents an optimization objective as a tree of variables that represent shard-to-server assignments. When evaluating a shard move, it only traverses tree nodes whose values may change, resulting in $O(\log(n))$ complexity. Finally, in addition to moving individual shards, it may consider two-way (or n-way) swapping of shards. ReBalancer has other advanced features to speed up search, which are beyond the scope of this paper.

Third, SM's allocator provides domain knowledge to guide ReBalancer to find a good solution quickly. It groups underutilized servers by properties (e.g., regions), samples servers from each group, and evaluates them as move targets. Unlike random sampling, sampling across groups has a better chance of finding a suitable move target for goals such as region preference and spread of replicas. This sampling approach helps reduce solving time significantly (§8.4). Moreover, SM's allocator groups placement goals of similar priorities into batches, and invokes ReBalancer on one batch at a time. Earlier batches focus on fixing the most critical violations (e.g., servers out of capacity) and can use search timeouts longer than later batches' timeouts. Within a batch, only the relevant goals are used to identify hot servers with relevant violations to fix, which simplifies the optimization problem in each batch and allows it to run faster. Finally, if a hot server has many small shards and a few large shards, going through the list sequentially may spend most of the time on evaluating moving small shards, which may be less effective in fixing violations. SM guides ReBalancer to evaluate large shards earlier, which not only accelerates the search but also reduces the number of shard assignment changes.

Overall, using a constraint solver is a major improvement over hand-crafted heuristics. It reduces the allocator's lines of code to ≈20% of the hand-crafted heuristics. For example, implementing spread of replicas takes only 180 lines of production code, including boilerplate code and monitoring instrumentation. ReBalancer's simple yet powerful APIs in Figure 13 enforce the separation of concerns. Systems experts focus on expressing placement problems, whereas optimization experts work on an abstraction that they are familiar with and can continuously improve the solver's performance and solution quality.

6 SM Scalability and Fault Tolerance

This section discusses the scalability and fault tolerance of the SM control plane.

6.1 SM's Scale-out Global Control Plane

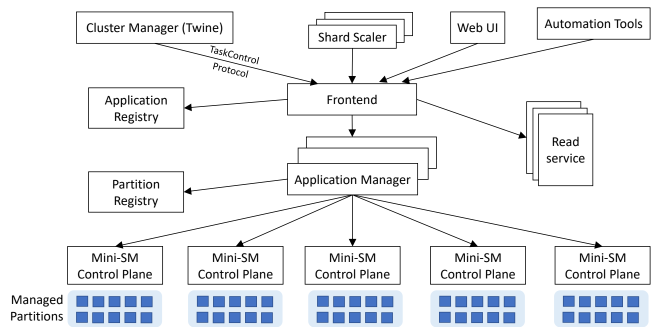

Figure 14. SM's scale-out global control plane.

The SM control plane in Figure 10 is not scalable enough to manage millions of servers and billions of shards. We divide SM's control plane into multiple *mini-SMs* so that each mini-SM manages a subset of servers and shards (Figure 14). In other words, we shard SM's control plane, but we use "*mini-SM*" to avoid confusion with application shards. Each mini-SM corresponds to the *SM Control Plane* in Figure 10.

A geo-distributed application can be so large that it is beyond what a mini-SM running on one machine can handle. We divide a large application into non-overlapping *partitions*, where each partition typically comprises thousands of servers and hundreds of thousands of shard replicas. Servers in a partition can come from different regions. Partitions are managed independently and can be assigned to different mini-SMs. The replicas of a shard are always placed on

servers that belong to the same partition. Because each partition is large, comprising thousands of servers, the average load in different partitions does not diverge much. In rare occasions that they do diverge, we use tools to initiate server or shard migration across partitions. As an application grows organically, new partitions can be added to scale it out.

Figure 14 shows SM's scale-out architecture. The *frontend* provides a global entry point. The *application registry* assigns applications to *application managers*. An application manager usually maps an application to one partition, but may divide a large application into multiple partitions. The *partition registry* assigns partitions to mini-SMs. A mini-SM can manage multiple partitions. As the system scales, more mini-SMs can be added to scale out. The *shard scaler* increases or decreases a shard's replica count in response to its load changes. The *read service* builds indices on mini-SM's metadata to serve queries.

6.2 Control Plane Fault Tolerance

SM is fault tolerant. Every component in Figure 14 has multiple replicas running in different regions. The frontend is stateless. Other components are stateful and use a primary-secondary setup. Every SM control-plane component in Figure 10 also has multiple replicas either running in the same region if it manages regional deployments, or running in different regions if it manages geo-distributed deployments.

We perform a staged rollout of a new release of the SM control plane over multiple days to prevent a bug from bringing down the entire SM control plane instantaneously. Even if all SM control-plane components are down, application clients can continue to send requests to application servers, although new shard assignments would not be generated.

7 Further Boosting SM's Adoption

Why adoption matters. In the first nine years of SM's history, its adoption was organic and rapid (Figure 2). In the past year, we switched from organic adoption to treating it as a priority to migrate legacy applications to SM because our infrastructure started to impose stronger contracts on applications. Without SM's help, each application would have to develop its own and often suboptimal solution to stay compliant with the following requirements.

Our infrastructure expects applications to improve resilience, agility, and efficiency, by 1) spreading a shard's replicas across large fault domains, 2) quickly draining shards out of a fault domain within a deadline, 3) dynamically adjusting shard placement as an auto-scaler adjusts the application's container count in response to load changes, and 4) maintaining high machine utilization via better LB and elimination of excessive regional deployments and excessive shard replicas.

Moreover, many applications use *taskIDs* for static sharding (§2.2.1), but sequential *taskIDs* are being deprecated. Burns et al. [11] called out that sequential *taskIDs* "*make it very hard to support jobs that span multiple clusters in a layer above (Google's) Borg*." We are rolling out a global federation layer atop Facebook's regional cluster managers (Twine [60]), which leads to the deprecation of sequential *taskIDs*.

The above forces that motivate the adoption of a sharding framework apply outside Facebook as well since the infrastructure contracts generally make applications better managed, and it has been a general trend to get rid of sequential *taskIDs* such as in Kubernetes.

Migrate legacy applications that use static sharding or consistent hashing. These applications account for 45% of Facebook's sharded applications (Figure 4). They often use extra hardware to compensate for the limitations of a simple sharding scheme (e.g., using excessive shard replicas to maintain availability) or simply bear with the risks (e.g., reduced availability in the event of a large fault-domain outage). However, these practices are problematic, and we expect most of these applications to migrate to SM in 1.5 years. Moreover, as the SM ecosystem is composable and supports incremental adoption of individual SM components, about 100 of these applications already adopted our generic shard TaskController [60] without using SM's APIs, allocator, or orchestrator. The generic shard TaskController uses an application-supplied shard map to decide whether certain container operations would endanger shard availability and instructs the cluster managers to operate accordingly.

Migrate legacy applications that use custom sharding. These applications account for only 1% of applications but 27% of server usage (Figure 4). Among them are Facebook's largest and most complex data stores [10, 43, 52]. It is impractical to fit them into SM's simple *add_shard()/drop_shard()* APIs. SM assumes that these APIs are implemented by the application servers, whereas these complex data stores have their custom central orchestrator to execute shard-assignment changes. For example, when adding a new database shard, their orchestrator may create both a database container and a sidecar container that does background maintenance and coordinates online schema changes.

These complex data stores still need to comply with the infrastructure contracts. SM's composable ecosystem allows them to benefit from SM's individual components without adopting the whole SM framework. Using Figure 10 as a reference, our strategy is to 1) allow them not to adopt SM's APIs, 2) keep their custom *orchestrator*, 3) develop their own *TaskController* based on the standard *TaskControl protocol*, 4) optionally reuse SM's *service-discovery system* to distribute their shard map, and 5) reuse a derived SM allocator called *Data Placer*. Data Placer saves significant efforts from the owners of these complex data stores as it can generate shard-to-server assignments that take into account both application-specific placement constraints and the infrastructure contracts. Most of the infrastructure contracts are related to shard-placement and load-balancing requirements, which are well handled by Data Placer. Data Placer is already

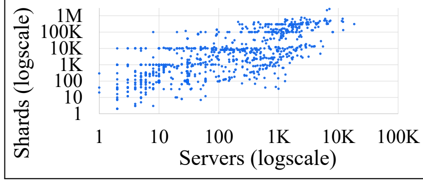

Figure 15. (Production) Scale of SM applications that run in production.

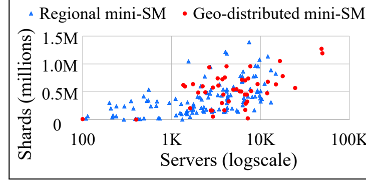

Figure 16. (Production) Scale of mini-SMs that run in production.

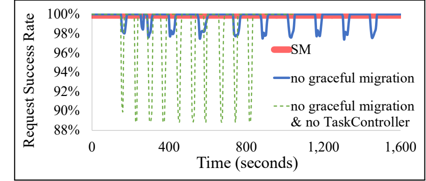

Figure 17. (Experiment) SM upholds availability during software upgrades.

integrated with these complex data stores and its production rollout is in progress.

8 Evaluation

We use both production data and experiments to answer the following questions:

- Overall, is SM scalable in production?
- Is SM effective in gracefully handling application upgrades to uphold application availability?
- Is SM effective in supporting geo-distributed applications?
- Is SM's constraint solver scalable and does it produce high-quality placement solutions?

8.1 Scale of SM in Production

Figure 15 shows the scale of SM applications, where each dot represents an application deployment that uses a certain number of servers to host a certain number of shards. The largest deployments use ≈19K servers and ≈2.6M shards. Most deployments are small, but 14% of the deployments use 1,000 or more servers. SM's scale-out architecture (Figure 14) is able to support many small and large applications, as it can add mini-SMs as needed. Currently, we operate 139 and 48 mini-SMs that manage regional and geo-distributed deployments (§2.2.2), respectively. Figure 16 shows the scale of these mini-SMs, where each dot represents a mini-SM that manages a certain number of servers and shards. The largest mini-SMs in production manage ≈50K servers and ≈1.3M shards. Each mini-SM runs on an 18-core 64GB RAM machine, with P90 CPU utilization at ≈30% and P90 memory utilization at ≈38%. In total, SM manages hundreds of applications' nearly 100M shards hosted on over one million servers, and those applications process billions of requests per second. SM gracefully handles millions of machine and network maintenance events per month.

8.2 Upholding Application Availability

Figure 17 evaluates how SM upholds application availability during application upgrades. We deploy a primary-only application with 10,000 shards on 60 servers. The application's configuration allows up to 10% of its containers to be restarted concurrently during a rolling upgrade. By gracefully handling container restarts (§4.1) and primary-replica migration (§4.3), the application client's request success rate stays at ≈100%. Without graceful primary-replica migration, the success rate drops to ≈ 98%. With neither graceful primary-replica migration nor graceful handling of container restarts, although the upgrade finishes earlier (800 seconds vs. 1,500 seconds), the success rate further drops below 90%.

Figure 18 uses production data to show that SM indeed upholds application availability. Facebook's instant-messaging product uses a queue service [22] to guarantee in-order message delivery to mobile devices. The service is a primary-only SM application. Its "*client request rate*" in Figure 18 follows a diurnal pattern. At peak, the service processes billions of requests per second. The service does a rolling upgrade every weekday. It starts with small-scale upgrades, which cause the small spikes in the "*shard moves*" curve, as SM gracefully migrates primary shards out of the impacted application servers. If no problem is detected, after three hours, it progresses to full-scale upgrades, which cause the big spikes in the "*shard moves*" curve. Despite the large number of concurrent shard moves, the "*client error rate*" curve hardly changes, showing that SM is effective in upholding the service's availability.

8.3 Supporting Geo-distributed Applications

In this experiment, we deploy a secondary-only application with 1,000 shards and two replicas per shard across three regions located at FRC (east coast of United States, Forest City, NC), PRN (west coast of United States, Prineville, OR) and ODN (Odense, Denmark), using 30 servers per region. Out of the 1,000 shards, 400 so-called east-coast (EC) shards are configured with a region preference for FRC because those shards are often accessed by east-coast users. In a steady state, each EC shard has one replica at FRC for locality and another replica at either PRN or ODN for fault tolerance.

Figure 19 shows the latency experienced by an application client located at FRC when it accesses the EC shards. In a steady state, it accesses the EC shards' replicas at FRC and the latency is low. At time 90 seconds, the servers at FRC fail and the client requests are routed to shard replicas at PRN or ODN, incurring a longer latency. The shard replicas originally at FRC also fail over to either PRN or ODN. The initial latency spike is due to request retries and requests bouncing between replicas at PRN and ODN until they stabilize on replicas that are closer. At time 450 seconds, the servers at FRC recover and SM migrates one replica of each EC shard back to FRC, bringing the client latency back to

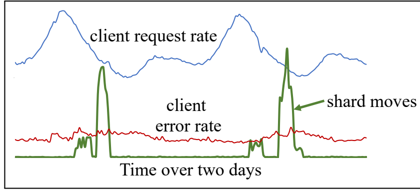

Figure 18. (Production) No increase in client errors during upgrades, thanks to graceful shard migration.

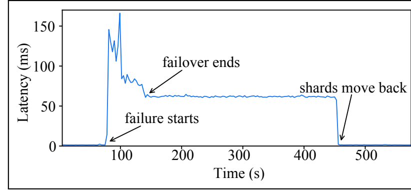

Figure 19. (Experiment) SM migrates a geo-distributed application's shards across regions to handle failures.

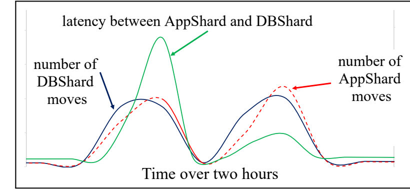

Figure 20. (Production) SM migrates AppShards across regions to follow DBShards to reduce network latency.

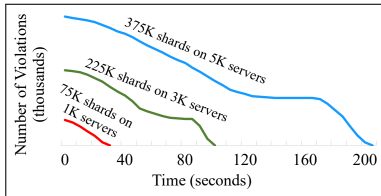

Figure 21. (Experiment) SM allocator scalability w.r.t. the problem size.

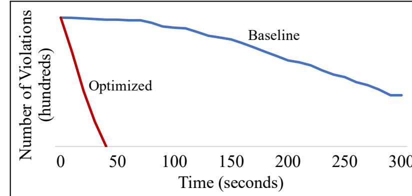

Figure 22. (Experiment) Optimizations help scale the constraint solver.

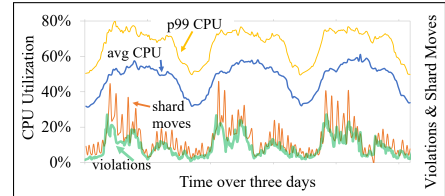

Figure 23. (Production) SM balances load in an ever-changing environment.

normal. In summary, this experiment showcases a typical example of how a geo-distributed application benefits from SM's support for region-aware shard placement and SM's ability to migrate shards across regions.

Figure 20 uses production data to demonstrate how SM-migrates shards across regions to reduce latency. Facebook's instant-messaging product stores messages in a sharded SQL database, which is not managed by SM. The application logic to process messages is implemented as an SM-based primary-only soft-state service (§2.4). All accesses to a given SQL database shard (so-called DBShard) must go through the same application shard (so-called AppShard). A pair of DB-Shard and AppShard should always run in the same region to minimize latency. In Figure 20, as part of a real production operation (i.e., not for the sake of collecting experimental data), an administrator initiates the first batch of DBShard moves across four regions, which causes a spike in latency because many pairs of DBShard and AppShard are separated into different regions. The administrator updates the regional placement preference for the impacted AppShards, which triggers SM to move the AppShards to co-locate with their DBShards. As a result, the latency goes back to normal. Half an hour later, the administrator initiates the second batch of DBShard moves and the process repeats again.

8.4 Scalability and Effectiveness of the SM Allocator

Figure 21 evaluates SM allocator's scalability. We take a snapshot of the server-capacity and shard-load information from a production deployment of ZippyDB [45]. SM balances load on three metrics: storage, CPU, and shard count. The shard load varies drastically—the largest shard's load is 20 times higher than that of the smallest shard. The server hardware is heterogeneous; e.g., the storage capacity varies by up to 20%. One load-balancing (LB) goal is to prevent a server's resource utilization from going above 90%; otherwise, it is considered a violation. Another LB goal is to cap the difference of server utilization within 10%; i.e., if a server's utilization exceeds the average utilization of all servers by 10%, it is also considered a violation.

Each experiment run's initial state starts with a random shard-to-server assignment in order to stress test the allocator with an unusually large number of violations to fix. We run the allocator at different scales and it is able to fix all violations in all stress tests, showing that it produces high-quality placement solutions. As the problem size grows 5x from 75K shards on 1K servers to 375K shards on 5K servers, the total solving time grows by 6.8x from 30 seconds to 205 seconds, indicating that the allocator is scalable. In our production environment, applications have much fewer violations than a random assignment and hence the solving time is much shorter, with P90 and P99 at ≈10 seconds and ≈50 seconds, respectively.

Figure 22 evaluates the effectiveness of optimization 4 described in §5.3, i.e., SM using domain knowledge to speed up local search. We compare the solving time for the 75K-shard problem with and without the optimization. Without the optimization, the allocator cannot even finish in 300 seconds and the resulting solution requires 22% more shard moves, which emphasizes the importance of the optimization.

Figure 23 uses production data to show how LB works in reality. It plots the CPU utilization, number of LB violations, and number of shard moves of a ZippyDB deployment, which all follow a diurnal pattern. As the number of violations in production is small, the SM allocator almost always

produces a new shard placement that can fix all existing violations. However, a small number of new violations constantly emerge on different servers due to the large system size (12K machines) and the ever-changing load driven by billions of Facebook product users' realtime activities. Despite the constant load changes, LB consistently keeps the P99 CPU utilization under 80%. This figure shows that LB is a continuous-optimization process as load constantly changes in production.

9 Related Work

Sharding frameworks. Slicer [4], Helix [27], Azure Service Fabric (ASF) [35], and Centrifuge [2] are closest to SM. ASF, Helix, Slicer, and SM all support some form of the primary and secondary roles. §3.1 compares the difference in sharding abstraction among Slicer, ASF, and SM. §2.4 further compares their difference in application-data persistency strategy. Our contributions listed in §1.2 distinguish SM from other sharding frameworks. Those advanced features have helped SM to gain a wide adoption (54%) that likely far exceeds what other sharding frameworks have achieved.

Application lifecycle management. Twine [60] allows an application to negotiate container starts or stops with the cluster manager. SM goes one step further to 1) gracefully handle primary-replica migration (§4.3), 2) globally coordinate container lifecycle operations across multiple geo-distributed cluster managers (§2.2.2), and 3) demonstrate that a common lifecycle manager (i.e., SM's TaskController) can be reused across hundreds of sharded applications. ASF "*provides full lifecycle management* [35]," but no information is available about whether or how ASF negotiates lifecycle events with the underlying data-center or physical-machine management layer. Regardless, since an ASF deployment cannot span across multiple clusters, it cannot coordinate lifecycle events across geo-distributed clusters.

Geo-distributed systems. SM draws inspiration from many geo-distributed systems [7, 14, 16, 40–42, 58, 65] and improves upon them by offering a generic framework to support diverse geo-distributed applications. Tuba [7] dynamically reconfigures the replica-placement policy for a specific geo-replicated storage system. Volley [5] automatically generates placement policies for geo-distributed cloud services based on request logs. SM provides a mechanism to enforce placement policies for diverse applications while upholding application availability. Currently, SM does not generate placement policies by itself, which is an active area of our ongoing work. At Facebook, Akkio [6] generates placement policies for micro shards and can be used along with SM. Kubernetes [37] can manage containers from a public cloud's geo-distributed regions, but cannot control the underlying cloud's VM lifecycle operations.

Using constraint solvers. Most cluster managers use handcrafted heuristics for resource allocation [4, 9, 13, 17–20, 32, 34, 37, 39, 60, 61, 63, 64, 66]. Systems using various solvers have been proposed [24, 26, 28–31, 33, 53, 57, 59, 62], but their production usage is rarely reported. Medea [24] uses a MIP solver to place long-running applications, and RAS [48] uses ReBalancer with a MIP solver to perform data-center resource allocation at Facebook. However, MIP is not scalable enough for near-realtime shard placement at a global scale. ASF attempted to use LP/IP and genetic algorithms, but found them not scalable or producing inferior solutions, and eventually adopted simulated annealing. Compared with simulated annealing, SM's local search employs advanced optimizations to speed up search (§5.3). SM can benefit from Wrasse's [53] approach of using GPUs to gain further speedup.

We scale a constraint solver beyond what was reported before and prove its practicality in production. The largest placement problem evaluated in DCM [57] involves 91K assignment variables, whereas the largest problem that SM solves in Figure 21 involves 1.9 billion variables. We solved many pending scalability issues reported in the DCM paper. For example, regarding "*it is likely overkill to evaluate every single node*", SM's allocator groups similar servers and samples them. Regarding "*this forces our generated code to construct models with redundant variables*," ReBalancer eliminates redundant variables by automatically identifying equivalent shards. DCM's declarative SQL interface is easier to use, whereas ReBalancer's imperative API interface allows using domain knowledge to speed up the solver.

10 Conclusion

We analyzed all sharded applications at Facebook and identified two barriers that fundamentally limit the adoption of existing sharding frameworks: 1) lack of support for geo-distributed applications and 2) inability to uphold application availability during planned events. To uphold availability, SM negotiates with the cluster managers about when to safely execute container lifecycle operations and also ensures that an application drops no request during a graceful shard migration. SM supports geo-distributed applications via region-aware shard placement and global coordination across regional cluster managers. We sped up a constraint solver to perform near-realtime shard placement at a global scale. Finally, SM is composable and allows applications to adopt its individual components to gain incremental benefits.

Our future work includes 1) accelerating the migration of legacy applications to SM (§7), 2) providing a smooth path to convert regional deployments to geo-distributed deployments (§2.2.2), and 3) managing an application's global-placement policy and capacity need, i.e., forecasting the number of servers needed for each region and placing shards intelligently to meet the application's global clients' latency requirements while minimizing the number of shard replicas.

References

[1] Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. 2003. *Local Search in Combinatorial Optimization.* Princeton University Press.

[2] Atul Adya, John Dunagan, and Alec Wolman. 2010. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. In *NSDI*, Vol. 10. 1–16.

[3] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-Value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems.* 113–119.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 739–753.

[5] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10).* USENIX Association, San Jose, CA. https://www.usenix.org/conference/nsdi10-0/volley-automated-data-placement-geo-distributed-cloud-services

[6] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 445–460.

[7] Masoud Saeida Ardekani and Douglas B Terry. 2014. A Self-configurable Geo-replicated Cloud Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 367–381.

[8] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10).* USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/finding-needle-haystack-facebooks-photo-storage

[9] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.*

[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference.* 49–60.

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (2016).

[12] Sergey Bykov, Alan Geller, Gabriel Kliot, Jim Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC 2011).*

[13] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. 2019. Hydra: A Federated Resource Manager for Data-center Scale Analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation.*

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[15] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data.* 1087–1098.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: GoogleâĂŹs Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles.*

[18] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based Scheduling: If You're Late Don't Blame Us!. In *Proceedings of the ACM Symposium on Cloud Computing.* 1–14.

[19] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[20] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[21] Jeffrey Dunn. 2016. Introducing FBLearner Flow: Facebook's AI backbone. https://engineering.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/.

[22] Jeremy Fein. 2014. Building Mobile-First Infrastructure for Messenger. https://engineering.fb.com/2014/10/09/production-engineering/building-mobile-first-infrastructure-for-messenger/.

[23] Fullmatix. 2014. https://github.com/kishoreg/fullmatix.

[24] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference.* 1–13.

[25] Gizzard. 2019. https://github.com/uber-node/ringpop-node.

[26] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N.M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.*

[27] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, Adam Silberstein, Kapil Surlaker, Ramesh Subramonian, and Bob Schulman. 2012. Untangling Cluster Management with Helix. In *Proceedings of the Third ACM Symposium on Cloud Computing.* 1–13.

[28] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[29] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-resource Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 65–80.

[30] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 81–97.

[31] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. 2012. VMware Distributed Resource Management: Design, Implementation, and Lessons Learned. *VMware Technical Journal* 1, 1 (2012), 45–64.

[32] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark

Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 845–861.

[33] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*.

[34] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.*

[35] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 33, 15 pages.

[36] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovic. 2019. Scribe: Transporting Petabytes per Hour via a Distributed, Buffered Queueing System. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/.

[37] Kubernetes. 2020. https://kubernetes.io/.

[38] Leslie Lamport et al. 2001. Paxos Made Simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[39] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Kunal Talwar, Udi Wieder, and Rama Ramasubramanian. 2011. *Validating Heuristics for Virtual Machines Consolidation.* Technical Report MSR-TR-2011-9.

[40] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–278.

[41] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 313–328.

[42] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.* 29, 4, Article 12 (Dec. 2011), 38 pages.

[43] Mark Marchukov. 2017. Facebook's Distributed Data Store for Logs. https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/.

[44] MarginSimulator. 2018. https://github.com/Dishan006/MarginSimulator.

[45] Sarang Masti. 2021. How We Built a General Purpose Key Value Store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-data/zippydb/.

[46] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. 2020. Turbine: FacebookâĂŹs Service Management Platform for Stream Processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1591–1602.

[47] Akshay Nanavati and Girish Joshi. 2021. FOQS: Scaling a Distributed Priority Queue. https://engineering.fb.com/2021/02/22/production-engineering/foqs-scaling-a-distributed-priority-queue/.

[48] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. 2021. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*.

[49] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christopher D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 33–46.

[50] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications.

[51] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1816–1827.

[52] Shlomo Priymak. 2013. Under the hood: MySQL Pool Scanner (MPS). https://engineering.fb.com/2013/10/22/core-data/under-the-hood-mysql-pool-scanner-mps/.

[53] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. 2012. Generalized Resource Allocation for the Cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–12.

[54] Ringpop. 2017. https://github.com/uber-node/ringpop-node.

[55] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 455–468.

[56] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 351–366. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/sharma

[57] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 827–844.

[58] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD'20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509.

[59] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. 2007. A Scalable Application Placement Controller for Enterprise Data Centers. In *Proceedings of the 16th international conference on World Wide Web*. 331–340.

[60] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor,

Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 787–803.

[61] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–14.

[62] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[63] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*.

[64] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[65] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. Spanstore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 292–308.

[66] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*.