Sharding the Shards: Managing Datastore Locality at Scale with Akkio

Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, and David Nagle, *Facebook;* Michael Stumm, *University of Toronto*

https://www.usenix.org/conference/osdi18/presentation/annamalai

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Sharding the Shards:
Managing Datastore Locality at Scale with Akkio

Muthukaruppan Annamalai,[†] Kaushik Ravichandran,[†] Harish Srinivas,[†] Igor Zinkovsky,[†]
Luning Pan,[†] Tony Savor,[†] David Nagle[†] and Michael Stumm[‡,†]

[†]Facebook, 1 Hacker Way, Menlo Park, CA USA 94025
{muthu,kaushikr,harishs,igorzi,luningp,tsavor,dfnagle}@fb.com

[‡]Dept. Electrical and Computer Engineering, University of Toronto, Canada M5S 3G4
stumm@eecg.toronto.edu

Abstract

Akkio is a locality management service layered between client applications and distributed datastore systems. It determines how and when to migrate data to reduce response times and resource usage. Akkio primarily targets multi-datacenter geo-distributed datastore systems. Its design was motivated by the observation that many of Facebook's frequently accessed datasets have low R/W ratios that are not well served by distributed caches or full replication. Akkio's unit of migration is called a *μ-shard*. Each μ-shard is designed to contain related data with some degree of access locality. At Facebook, μ-shards have become a first-class abstraction.

Akkio went into production at Facebook in 2014, and it currently manages ∼100PB of data. Measurements from our production environment show that Akkio reduces access latencies by up to 50%, cross-datacenter traffic by up to 50%, and storage footprint by up to 40% compared to reasonable alternatives. Akkio is scalable: it can support trillions of μ-shards and process many 10's of millions of data access requests per second. And it is portable: it currently supports five datastore systems.

1 Introduction

This paper regards the management of data access locality in large distributed datastore systems. Our work in this area was initially motivated by our aim to reduce service response times and resource usage in our cloud environment which operates globally and at scale: the computing and storage resources are located in multiple geo-distributed datacenters, hundreds of petabytes of data must be available for access, data accesses occur at the rate of many tens of millions per second, and the location from which any data item is accessed changes dynamically over time. Many organizations are increasingly faced with some, if not all, of these aspects, as they target a growing user base around the world. Indeed, geo-distributed systems are becoming increasingly prevalent and important, as witnessed by Spanner Cloud and CockroachDB, two cloud-based geo-distributed datastore systems available to any organization [17, 38].

Managing data access locality[1] in geo-distributed systems is important because doing so can significantly improve data access latencies, given that intra-datacenter communication latencies are two orders of magnitude smaller than cross-datacenter communication latencies; e.g., 1ms vs. 100ms. Locality management can also significantly reduce cross-datacenter bandwidth usage, which is important because the bandwidth available between datacenters is often limited (§2.1), potentially leading to communication bottlenecks and attendantly higher communication latencies. Managing locality is all the more challenging when considering that access patterns can change geographically over time; particularly, when shifting workload from one datacenter operating at high utilization (e.g., during its day) to another operating at low utilization (e.g., its night) (§2.2).

We argue that explicit data migration is a necessary mechanism for managing data access locality in geo-distributed environments, because existing alternatives have serious drawbacks in many scenarios. For instance, distributed caches can be used to improve data read access locality. However, because misses often incur remote communications, these caches require extremely high cache hit rates to be effective, thus demanding significant hardware infrastructure. Further, distributed caches do not typically offer strong consistency (§2.4). Another alternative is to fully replicate data with a copy in each datacenter to allow for (fast) localized read accesses. However, as the number of datacenters increases, storage overhead becomes exorbitant with large amounts of data, and also write overheads increase significantly, as all replicas need to be updated on each write (§2.1). At Facebook, many of the heavily accessed datasets have

[1] Our use of the term *locality* should not be confounded with the term *localization*; the solution we propose here is not suitable for segregating data by region.

relatively low read-write ratios (§2.3), so full replication would consume excessive cross-datacenter bandwidth. A third alternative is function shipping. But this can also be ineffective, as it may still result in significant cross-datacenter communications, the target datacenter may be operating at peak capacity, or the required data may be located in multiple datacenters.

Akkio. In this paper we present ***Akkio***,[2] a locality management service for distributed datastore systems whose aim is to improve data access response times and to reduce cross-datacenter bandwidth usage as well as the total amount of storage capacity needed. Akkio is layered between client applications servicing client requests and the distributed datastore systems used natively by the client applications. It decides in which datacenter to place and how and when to migrate data, and it does so in a way that is transparent to its clients and the underlying datastore system.[3] It helps direct each data access to where the target data is located, and it tracks each access to be able to make appropriate placement decisions. Akkio has been in production use at Facebook since 2014 and thus operates at scale: it currently manages over 100PB of data and processes many tens of millions of data accesses per second (despite Akkio not being suitable many of Facebook's datasets).

μ-shards. Having migration as the basis for providing data access locality raises the question: what is the right granularity for migrating data? A ubiquitous method in distributed datastore systems is to partition the data into *shards* using key ranges or key hashing [26, 37]. Shards serve as the unit for replication, failure recovery, and load balancing (e.g., upon detection of query or storage load imbalances, shards are migrated from one node to another to rebalance the load). Each shard is on the order of one to a few tens of gigabytes, is assigned in its entirety to a node, and multiple shards (10s – 100s) are assigned to a node. Shard sizes are set by the datastore administrator to balance (*i*) the amount of metadata needed to manage the shards with (*ii*) effectiveness in load balancing and failure recovery (§2.5). Notably, datastore systems define shards in an application-transparent manner.

Given the ubiquity of shards, migrating data at shard granularity is an option; in fact, a few systems that do this have been proposed [4, 12, 29, 40]. However, this approach has a serious drawback given typical shard sizes: the vast majority of the migrated data would likely not belong to the working set of the accessing workload at the new location, thus incurring unnecessary migration overhead and wasting inter-datacenter WAN communication bandwidth. At Facebook, because the working set size of accessed data tends to be less than 1MB, migrating an entire shard (1-10GB) would be ineffective.

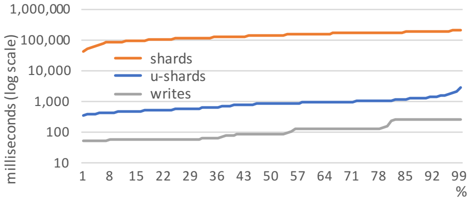

Figure 1: Cumulative distribution of cross-datacenter transfer times. Each curve contains data obtained from 10,000 randomly sampled data points across all cross-datacenter links at Facebook. Avg. shard size is 2GB; avg. μ-shard size is 200KB.

In this paper, by way of Akkio, we advocate for the notion of finer-grained datasets to serve as the unit of migration when managing locality. We call these finer-grained datasets ***μ-shards***. Each μ-shard is defined to contain related data that exhibits some degree of access locality with client applications. It is the application that determines which data is assigned to which μ-shard. At Facebook, μ-shard sizes typically vary from a few hundred bytes to a few megabytes in size, and a μ-shard (typically) contains multiple key-value pairs or database table rows. Each μ-shard is assigned (by Akkio) to a unique shard in that a μ-shard never spans multiple shards.

μ-shards are motivated by our observation that there exist datasets that exhibit good access locality with respect to a client application, but that they are best identified by the client application. Hence, μ-shards are not simply smaller-sized shards. The primary difference between shards and μ-shards, besides size, is the way data is assigned to them. With the former, data is assigned to shards by key partitioning or hashing with little expectation of access locality. With the later, the application assigns data to μ-shards with high expectation of access locality. As a result, μ-shard migration has an overhead that is an order of magnitude lower than that of shard migration (Fig. 1), and its utility is far higher.

μ-shards offer their best advantages in contexts where it is unambiguous how to set the unit of migration so that it is simultaneously as large as possible, meets the constraints of good access locality, and primarily contains data belonging to the same working set of an accessing workload. We have found that there exist many datasets where these parameters are easily identified; see Table 1 for some examples. Because of this, we argue it is propitious to make μ-shards a first class abstraction, such that they are visible to and specified by client applications. The motivation is that only the client applications have the domain knowledge to best determine which data are related and likely to be used together.

Having the application identify related data is not an

[2] Akkio is a play on Harry Potter's Accio Summoning Charm that summons an object to the caster, potentially over a significant distance [31].

[3] In this paper we use the term "underlying datastore system" to refer to the datastore system used natively by the client application. It may be different than the datastore system used by Akkio.

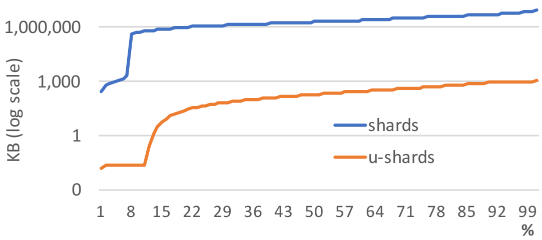

Figure 2: Cumulative distribution of Shard and µ-shard size for ViewState datasets. The ViewState service keeps track of content previously shown to the end-user. ViewState µ-shard sizes tend to be larger than the size of the typical µ-shards managed by Akkio (500KB avg. vs. 200KB avg.).

unreasonable expectation. Many applications already group together data by prefixing keys with a common identifier to ensure that related data are assigned to the same shard. This approach has been used for a long time in practice. Similarly, some databases support the concept of separate partition keys. Spanner supports "directories" although Spanner may shard directories into multiple fragments [11]. Finally, a number of Facebook-internally developed databases, including ZippyDB, support µ-shards as a first class abstraction in the sense that each access request also includes a µ-shard id [3, 8, 34].

Akkio's functionality. Akkio is implemented as a layer between client applications and the underlying datastore system that implements sharding. Although µ-shards are defined by the client applications, Akkio manages them in an application-transparent manner. Akkio is responsible for: (i) tracking client-application accesses to µ-shards so it can take access history into account in its decision making; (ii) deciding where to place each µ-shard; (iii) migrating µ-shards according to a given migration policy for the purpose of reducing access latencies and WAN communication; and (iv) directing each access request to the appropriate µ-shard. Akkio takes capacity constraints and resource loads into account in its placement and migration decisions, even in the face of a heterogeneous environment with a constantly churning hardware fleet.

Akkio is able to support a variety of *replication configurations* and *consistency* requirements (including strong consistency) as specified by each client application service. This flexibility is provided because the client application service owners are in the best position to make the right tradeoffs between availability, consistency, resource cost-effectiveness, and performance. Akkio maps each µ-shard with a specified replication requirement onto a shard configured with the same replication and consistency requirements in the underlying datastore system. As well, it enforces the specified level of consistency during µ-shard migrations.

- web application user profile information
- Amazon user browsing history to inform recommendations
- Spotify user listening history to inform subsequent content
- Facebook viewing history to inform subsequent content
- Slack group recent messages
- Reddit subreddits
- email folders
- messaging queues

Table 1: Example datasets conducive to µ-shards. Note that all but the first exhibit relatively low read-write ratios.

Other applications. While this paper focuses on Akkio managing locality for geo-distributed environments, Akkio and its mechanisms can be useful in other scenarios. For example, Akkio can be used to migrate µ-shards between cold storage media (e.g. HDDs) and hot storage media (e.g., SSDs) on changes in data temperatures, similar in spirit to CockroachDB's archival partitioning support [38]. Further, for public cloud solutions, Akkio could migrate µ-shards when shifting application workloads from one cloud provider to another cloud provider that is operationally less expensive [39]. Finally, when resharding is required, Akkio could migrate µ-shards, on first access, to newly instantiated shards, allowing a more gentle, incremental form of resharding in situations where many new nodes (e.g. a row of racks) come online simultaneously.

Contributions. We describe the design and implementation of Akkio (§4). To the best of our knowledge, Akkio is the first system capable of managing data locality at µ-shard granularity and at scale, while also supporting strong consistency. In describing Akkio, we focus on scalability; in that sense, this paper focuses on the "plumbing" and not on policy; i.e., specific decision-making algorithms. For applications where Akkio is suitable, we show in §5 that Akkio is:

Effective along a number of dimensions: Compared to typical alternatives, Akkio can achieve *read latency reductions*: up to 50%; *Write latency reductions*: 50% and more; *Cross-datacenter traffic reductions*: by up to 50%. Further, Akkio reduces storage space requirements by up to $X-R$ compared to full replication with X datacenters when a replication factor of R is required for availability.

Scalable: Statistics from production workloads servicing well over a billion users demonstrate the system remains efficient and effective even when processing many tens of millions of requests per second. Akkio can support trillions of µ-shards.

Portable: Akkio's design is simple and flexible enough to allow it to be easily layered on top of most backend datastore systems. Akkio currently runs on top of ZippyDB, Cassandra, and three other internally-developed databases at Facebook.

Limitations. Akkio's approach to managing locality with µ-shards will not be beneficial for all types of data, such as those better served by distributed caches, or

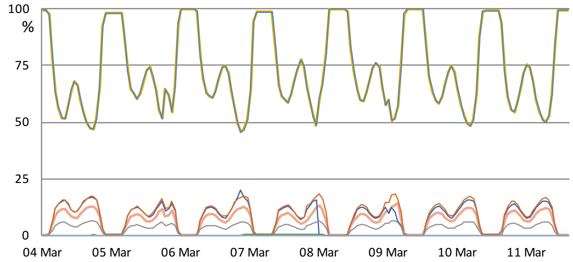

Figure 3: Proportion (in %) of incoming service requests originating from Region A processed at each datacenter. Each curve represents a datacenter. The sum over all curves is always equal to 100%. In this case, Region A has a local datacenter.

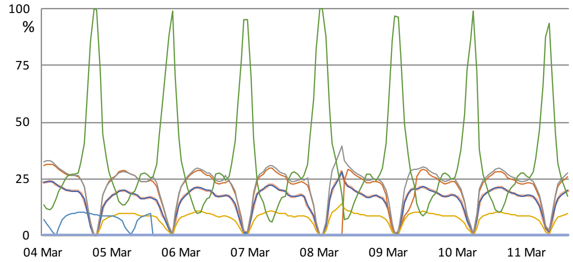

Figure 4: Proportion (in %) of incoming service requests originating from Region B processed at each datacenter. In this case, Region B does not have a local datacenter.

datasets that do not exhibit sufficient access locality. For example, Akkio would not helpful in improving locality for data belonging to the Social Graph. Instead Akkio focuses on workloads with datasets that have low read-write ratios and high access locality. These workloads are quite common and not well served by a caching tier. Further, while Akkio can be layered on top of a variety of datastores, the datastore needs to provide particular features to Akkio as outlined in §4.2. As a result, Akkio may not be able to accommodate all datastore systems. Finally, Akkio does not currently support inter-μ-shard transactions, unless implemented entirely client-side; providing this support is left for future work.

We begin the paper by substantiating our motivation underlying Akkio's approach (§2) and present background needed to understand the rest of the paper (§3).

2 Motivation

2.1 Capital and operational costs matter

Capital and operational costs become consequential when an organization's infrastructure must scale to target a large number of users around the world, justifying considerable efforts to restrain resource usage where possible. Consider an organization with ten datacenters and many hundreds of petabytes of data that must be accessible. While it is difficult to obtain transparent, publicly available pricing information on the true cost of storage, a lower bound for capital depreciation and operational costs could be on the order of two cents per gigabyte per month [9, 28]. This translates to $2 million per 100 petabytes per month. Clearly, replicating all data onto all ten datacenters is difficult to justify from an economic perspective when, in many cases, acceptable durability could be achieved with three replicas.

WAN cross-datacenter links can also be costly and need to be taken into account. For example, estimates for the costs of a 10 Gbps subterranean link vary from $1 to $9 per km per month, depending on route [22]. (To put this into perspective, transferring a 10 GB shard over a 10 Gbps WAN link will consume roughly 10 seconds of bandwidth.) As a result, cross-datacenter link bandwidth will typically be constrained and therefore needs to be used judiciously.

2.2 Service request movements

The datacenters from which data access requests originate can vary over time, even for data accessed on behalf of a unique user. A change in the requesting datacenter can arise, for example, because the user travels from one region to another, or, more likely, because service workload is shifted from a datacenter with high loads to another with lower loads in order to lower service request response latencies. The alternative to shifting workload to other datacenters at peak times would be to increase the capacity of the overloaded datacenter to deal with peak influx of service requests. But this comes with significant operational overheads, which are hard to justify when other datacenters are mostly idle at the same time, given diurnal request patterns.

Figure 3 shows that shifts in traffic occur on a daily basis at Facebook. The figure shows which datacenters processed incoming service requests originating from one particular region over a week. Each curve represents a different datacenter to which the service requests originating from one region were forwarded. The figure shows that during busy periods, as many as 50% of the requests originating from the given region were shifted to remote datacenters (most often located in an adjacent region). The figure also shows that during non-peak times all of the requests are processed by the local datacenter.

Figure 4 shows the same type of information, but for a region with no local datacenter. Because there is no local datacenter, the service requests are distributed to a number of different datacenters. During non-peak times, we see that almost all traffic is serviced from a single, non-local, but nearby datacenter.

We also measured, for each individual end-user, how many datacenters processed service requests issued on behalf of that user over a period of a week (Table 2): over

Num regions:	1	2	3	4
% of users:	46%	42%	10%	2%

Table 2: The percentage of users for which Num regions were contacted to service requests on behalf of the user.

54% of users have their data accessed from two or more regions. Bottom line: there is a reasonable likelihood that requests issued on behalf of one end-user will be processed by multiple distinct datacenters.

2.3 Low read-write ratios

Many important datasets exhibit low read-write ratios (Table 1). As a Facebook-specific example, dataset ViewState (§5.2.1) keeps track of content previously shown to the end-user and has a read-write ratio of 1. Overall, Facebook has on the order of 100PB of periodically accessed data that has a read-write ratio below 5. Note that with low read-write ratios, fully-replicated data would incur significant cross-datacenter communication, as all replicas would have to be updated on writes.

2.4 Ineffectiveness of distributed caches

A common strategy to obtain localized data accesses is to deploy a distributed cache at each datacenter [2, 5, 13, 14, 15, 27, 32]. In practice this alternative is ineffective for most of the workloads important to Facebook. First, unless the cache hit rate in the cache is extremely high, average read latencies will be high if the target data is not located in the local datacenter. Because of this, caching will demand significant hardware infrastructure, as the caches at each datacenter would have to be large enough to hold the working set of the data being accessed from the datacenter.

Second, low read-write ratios lead to excessive communication over cross-datacenter links, because the data being written will, in the common case, be remote.

Finally, many of the datasets accessed by our services require strong consistency. While providing strongly consistent caches is possible, it significantly increases the complexity of the solution, and it incurs a large amount of extra cross-datacenter communication, further exacerbating WAN latency overheads. It is notable that the widely popular distributed caching systems that are scalable, such as Memcached or Redis, do not offer strong consistency. And for good reason.

2.5 Separate locality management layer

Akkio is implemented as a layer between the application service and the underlying distributed datastore system. This raises the question of whether it would make more sense to implement Akkio's functionality directly within the datastore system. Technically, it would be possible, but we argue that this is not a good idea for two reasons.

First, the size of shards are carefully selected by the datastore architects for the purpose of managing load balancing and failure recovery, taking into account the configuration and other metadata needed to manage the shards. Maintaining this data at µ-shard cardinality would come at high storage overheads with 100's of billions of µ-shards vs. 10,000's of shards. Restructuring a datastore system to achieve the level of scale required to support µ-shards across each of its layers would require non-trivial changes.

Second, many application services use data that are not well-served by Akkio-style locality management; e.g., Google search or Facebook's social graph. Hence, it would only make sense to incorporate Akkio's functionality into *specialized* datastore systems; given that datastore system designers optimize for the common case, they would be reluctant to incorporate the additional complexities associated with µ-shards. However, even with a specialized datastore system, legacy issues come into play; in our experiences, application service owners are reluctant to switch away from the underlying datastore system for which their service was tuned and on which they rely for special features or behaviors.

We believe that Akkio bridges the functionality offered by various distributed datastore systems and the application services' desire for (transparent) data locality management to improve response times and reduce WAN datalink overheads.

3 Background

In this section, we briefly review several aspects of shard replication in distributed datastore systems so we can explain Akkio's architecture in §4. In doing so, we introduce some vocabulary we use in subsequent sections. Without loss of generality, we specifically describe how shard replication is handled in ZippyDB, an internally developed scalable key-value store system.[4]

ZippyDB's data is partitioned horizontally, with each partition assigned to a different shard. Each shard may be configured to have multiple replicas, with one designated to be the *primary* and the others referred to as *secondaries*. (See Fig. 5.) We refer to all of the replicas of a shard as a ***shard replica set***, and each replica participates in a shard-specific Paxos group [21, 24, 25]. A write to a shard is directed to its primary replica, which then replicates the write to the secondary replicas, using Paxos to ensure that writes are processed in the same order at each

[4] ZippyDB is used as the database service for hundreds of use cases at Facebook including news products, Instagram services and WhatsApp components. An increasing number of services are being moved onto ZippyDB at Facebook.

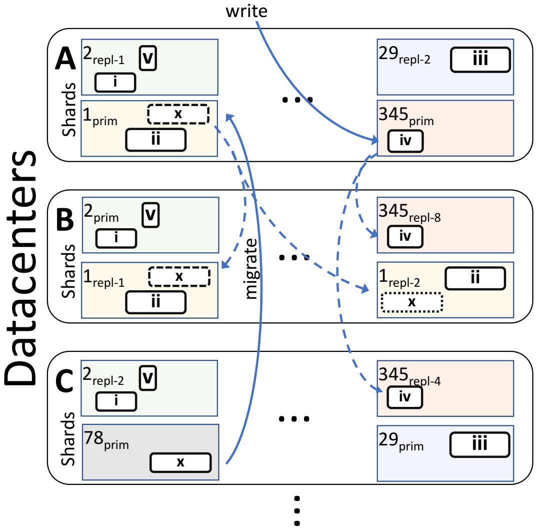

Figure 5: Shards with different replication configurations distributed across datacenters. The shaded rectangles represent shards. Shard 1 has the primary replica in Datacenter A and two secondary replicas in Datacenter B. Shard 2 is replicated across A, B, and C with the primary in B. The smaller boxes represent µ-shards. µ-shard v is assigned to replica set 2; a write that modifies µ-shard v is directed to replica set 2's primary replica and the underlying datastore system replicates the write to the secondary replicas. Akkio is migrating µ-shard x from replica set 78 to replica set 1, and the datastore system replicates x onto 1's secondary.

replica. Reads that need to be strongly consistent are directed to the primary replica. If eventual consistency is acceptable then reads can be directed to a secondary.

A shard's ***replication configuration*** identifies the number of replicas of the shard and how the replicas are distributed over datacenters, clusters, and racks. Shard replication configurations are customizable given that the data owners are in the best position to make the right tradeoffs between availability, consistency, resource-effectiveness, and performance. For example, a service may specify that it requires three replicas, with two replicas (representing a quorum) in one datacenter for improved write latencies and a third in different datacenter for durability. Another service may specify that it requires three replicas located in three different datacenters but that eventual consistency is sufficient. A third might require only one copy, perhaps because the infrastructure overhead of having multiple copies may be deemed to be too high relative to the value of the data.

We use the term ***replica set collection*** to refer to the group of all replica sets that have the same replication configuration. Each such collection is assigned a unique id we call a ***location handle***. When running on top of ZippyDB, Akkio places µ-shards on, and migrates µ-shards between different such replica set collections.

Fig. 5 depicts several shard replica sets and a number of µ-shards within the replica sets. It also shows how a write to a µ-shard is propagated to all secondaries.

Replica sets collections are provisioned and made available to a client application service by a utility that takes input from the client application service owners to help them make the right tradeoffs between availability, consistency, resource-effectiveness, and performance. For example, it inputs application-service parameters that include expected data size, expected access rate (i.e., QPS), R/W-ratios, etc. It also inputs policy parameters that include replication factor, availability requirements, consistency requirements and constraints with respect to where the replicas can be placed.

In general, all possible configurations are included that minimize the replication factor (within the specified constraints). However, some configurations may be excluded. For example, for ViewState, all replica set configurations with three replicas in three different datacenters are excluded so that two replicas will always be located in the same datacenter so that writes have lower latency (given the applications low R/W-ratio).

Once shards have been provisioned, then ZippyDB's *Shard Manager* assigns each shard replica to a specific ZippyDB server while obeying the specified policy rules. The assignment is registered with a *Directory Service* so that the ZippyDB client library embedded in the application service can identify the server to send its access requests to. Shard Manager is also responsible for: (*i*) load balancing, by migrating shards if necessary; and (*ii*) monitoring the liveliness of ZippyDB servers, taking appropriate action when a server failure is detected.

As a final comment, we observe that ZippyDB is able to manage multiple different replication configurations inside a single ZippyDB deployment. Other datastore systems may not be able to support multiple configurations inside a *single* deployment. However, in that case one can usually implement different replication configurations in a straight-forward way by using *multiple* datastore deployments.

4 Akkio Design and Implementation

For clarity, we describe Akkio's design and implementation in the context of a single client application service, ViewState, which uses ZippyDB as its its underlying datastore system. This is without loss of generality, because the underlying database is unaware of Akkio's presence beyond a small portion of code in the database client library.

4.1 Design guidelines

Akkio's design is informed by three primary guidelines. First, Akkio uses an additional level of indirection: it maps µ-shards onto shard replica set collections whose shards are in turn mapped to datastore storage servers. This allows Akkio to rely on ZippyDB functionality to provide replication, consistency, and intra-cluster load balancing. Secondly, Akkio is structured so as to keep most operations asynchronous and not on any critical path — the only operation in the critical path is the µ-shard location lookup needed for each data access to identify in which replica set collection the target µ-shard is located. Thirdly, Akkio minimizes the intersection with the underlying application datastore tier (e.g., ZippyDB), which makes it more portable. The only two points where the datastore system and Akkio meet are in the datastore client libraries and in Akkio's migration logic which is specific to the datastore.

4.2 Requirements

Akkio imposes three requirements on client application services that wish to use it. First, the client application service must partition data into µ-shards, which are expected to exhibit a fair degree of access locality for Akkio to be effective. Second, the client application service must establish its own µ-shard-id scheme that identifies its µ-shards. µ-shard-ids can be any arbitrary string, but must be globally unique. Finally, to access data in the underlying application database, the client application service must specify the µ-shard the data belongs to in the call to the database client library. For databases that do not support µ-shards natively as ZippyDB does, the function used to access data is modified to include a µ-shard-id as an argument to each access request; e.g., `read(key)` must be modified to `read(µ-shard-id,key)`.

Akkio imposes two requirements on the underlying database. First, the database must ensure µ-shards do not span shards. Because ZippyDB understands the notion of µ-shards, it will never partition µ-shards. Many databases support explicit partition keys that inform the database how to partition data (e.g., MySQL, Cassandra). Yet other databases may recognize key prefixes when partitioning data (e.g., HBase, CockroachDB).

Second, the underlying application database must provide a minimal amount of support so that Akkio can implement migration while maintaining strong consistency. Because the specific features supported by different datastore systems will vary, the µ-shard migration logic that Akkio implements must be specific to the underlying datastore system being supported. For example, some databases, including ZippyDB, offer access control lists (ACLs) and transactions, which are sufficient for implementing µ-shard migration. Other databases, including Cassandra, offer timestamp support for ordering writes, which is also sufficient.

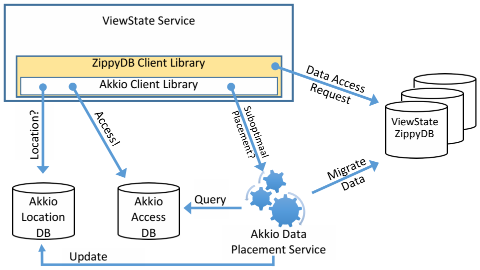

Figure 6: Akkio System Design

4.3 Architectural overview

Akkio's general architecture is depicted in Figure 6. A portion of Akkio's logic is located in the Akkio Client Library, which is embedded into the database client library; i.e., ZippyDB client library, in this case. The client application service makes data access requests by calling the ZippyDB client library, which in turn may make calls to the Akkio Client Library. Beyond the Akkio Client Library, Akkio is made up of three services, which are depicted at the bottom of the figure and described in more detail in the subsections that follow.

The **Akkio Location Service** (ALS) maintains a *location database*. The location database is used on each data access to look up the location of the target µ-shard: the ZippyDB client library makes a call to the Akkio client library `getLocation(µ-shard-id)` function which returns a ZippyDB *location handle* (representing a replica set collection) obtained from the location database. The location handle enables ZippyDB's client library to direct the access request to the appropriate storage server. The location database is updated when a µ-shard is migrated.

An **Access Counter Service** (ACS) maintains an access counter database, which is used to track all accesses so that proper µ-shard placement and migration decisions can be made. Each time the client service accesses a µ-shard, the Akkio client library requests the ACS to record the access, the type of access, and the location from which the access was made. This request is issued asynchronously so that it is not in the critical path.

The ACS is primarily used by Akkio's third service, the **Data Placement Service** (DPS), which decides where to place each µ-shard so as to minimize access latencies and reduce resource usage. The DPS also initiates and manages µ-shard migrations. The Akkio Client Library asynchronously notifies the DPS that a µ-shard placement may be suboptimal whenever a data access request needs to be directed to a remote datacenter. The DPS re-evaluates the placement of a µ-shard only when it receives such a notification. This ensures the DPS triggers migrations only when needed, thus effectively prioritizing migrations and preventing unnecessary migrations for µ-shards that are not being accessed. Note, however, that a µ-shard access never waits for a potential migration to be evaluated or complete, but proceeds directly with the remote access.

We now discuss these services in more detail.

4.4 Akkio Location Service (ALS)

The Akkio Location Service maintains a database that stores the location handle of each µ-shard. In principle, most any database could be used for storing this information; here we use ZippyDB (without Akkio layered on top of it).[5] The location information is configured to have an eventually consistent replica at every datacenter to ensure low read latencies and high availability, with the primary replicas evenly distributed across all datacenters. This configuration is justified, given the high read-write ratio (> 500) of the database. Moreover, distributed in-memory caches are used at every datacenter to cache the location information so as to reduce the read load on the database, considering that the database needs to be queried on every access request.

It is possible that the distributed cache will serve a stale location mapping, causing the access request to be sent to the wrong server. The target ZippyDB server will determine that the µ-shard is not present from the missing ACL, and will respond accordingly. When that happens, the ZippyDB client library queries the Akkio Location Service again, this time requesting that the cache be bypassed. The client library subsequently re-populates the cache with the latest mapping (making the cache a typical demand-filled look aside cache).

The amount of storage space needed for the ALS is relatively small: each µ-shard requires at most a few hundred bytes of storage, so the size of the dataset for typical client application services will be a few hundred GB. The overhead of maintaining a database for this amount of data in every datacenter is trivial. Similarly, the in-memory caches require no more than a handful of servers per datacenter, since a single machine can service millions of requests per second. The service can easily scale by increasing the number of caching servers.

4.5 Access Counter Service

Access counters are used to keep track of where µ-shards are accessed from and how frequently. To maintain this information, we use the time-windowed counters [7] provided natively by ZippyDB. The counter database uses a separate, dedicated ZippyDB instance, configured to use 3X replication. For each client application service, Akkio stores a single counter per µ-shard per datacenter.

```
consistency_requirements = STRONG;
replication_configurations = {
    ”location_handle_a”: <A, B, C>
    ”location_handle_b”: <D, E, F>
    ....
};
access_counter_service = AccessState;
migration_policy = MigrationPolicy(
    microshard_limit=6hours);
```

Listing 1: Akkio Configuration for Sample Service

The amount of storage needed for the counters is on the order of 10's of bytes per µ-shard and datacenter; in our environment less than 200GB per datacenter, which is again trivial. The counter service can easily scale by spreading the counters over a larger number of servers. As an optimization, the number of counters needed and the overhead of incrementing them can be reduced substantially by observing that many of the client application services have identical access patterns. For example, Facebook's AccessState service, which records actions taken in relation to displayed content, has very similar access traffic patterns as ViewState, which records which content was displayed; the traffic of both services is driven by Facebook user traffic. For this reason, Akkio allows a client application service to specify that the counters of another service should be used as a proxy for its own access pattern, in which case the application service does not need a separate set of counters. Moreover, the requests are batched and send-optimized, so the extra communication traffic generated is marginal. (With our workload, ACS adds 0.001% in networking bandwidth.)

4.6 Akkio Data Placement Service (DPS)

Akkio's Data Placement Service is responsible for mapping µ-shards to location handles and for migrating µ-shards in order to improve locality. There is one DPS per Akkio-supported backend datastore system that is shared among all of the application services using instances of that same datastore system. It is implemented as a distributed service with a presence in every datacenter.

The two main interfaces exported by DPS are `createUshard()` and `evaluatePlacement()`. New µ-shards are provisioned on demand when a µ-shard is accessed for the first time; in that case, the Akkio client library receives an `UNKNOWN_ID` response from ALS, so it invokes `createUshard()` (§4.6.1). `EvaluatePlacement()` is invoked by the Akkio client library asynchronously. It first checks whether initiating a migration is permissible, by checking whether the policy allows the target µ-shard to be migrated at that time, and whether the µ-shard is not already in the process of being

[5] If the application service uses a different underlying datastore system, we use a separate instance of that datastore system for the location database. We do this because the product owners of the underlying datastores were hesitant to allow another system to be in the critical path of data accesses to their system. The two other Akkio services use ZippyDB regardless.

migrated. If migration is permissible, it determines the optimal placement for the µ-shard (§4.6.2) and starts the migration (§4.6.3).

DPS stores various information in its datastore system for each µ-shard migration, including locks to prevent multiple concurrent migrations of the same µ-shard, and sufficient information needed to recover the migration should a DPS server fail during the migration (e.g., from and to location handles, lock owners, etc). As well it maintains historical migration data: e.g., time of last migration to limit migration frequency (to allow the prevention of µ-shards ping-ponging).

4.6.1 Provisioning new µ-shards

When a new µ-shard is being created, DPS must decide where to initially place the µ-shard. Our typical strategy is to select a replica set collection with a primary replica local to the requesting client and secondary replica(s) in one of the more lightly loaded datacenters. But, in principle, any available shard replica set collection could be chosen, so using a hash function to distribute initial µ-shard assignments is also a viable strategy.

The primary reason µ-shard provisioning is delegated to DPS is that if any Akkio client library instance were to do this directly, then a race condition might ensue if two or more client instances decide to create the same new µ-shard concurrently. A further advantage of leveraging DPS is that current resource usage can be taken into account when placing the µ-shard.

4.6.2 Determining optimal µ-shard placement

The default policy for selecting a target replica set collection for an existing µ-shard is to choose the one with the highest *score* from among the available replica set collections, excluding those with replicas in datacenters with exceptionally high disk usage or exceptionally high computing loads.[6] Our implementation computes the score in two steps. First, we compute a per-datacenter score by summing the number of times the µ-shard was accessed from that datacenter over the last X days (where X is configurable), weighting more recent accesses more strongly. The per-datacenter scores for the datacenters on which the replica set collection has replicas are then summed to generate a replica set collection score. If there is a clear winner, we pick that winner.

```
Atomically:
  a. acquire lock on u-shard
  b. add migration to ongoing migrations list
Set src u-shard ACL to R/O;
Read u-shard from the src
Atomically:
  - write u-shard to dest
  - set dest u-shard ACL to R/O
Update location-DB with new u-shard mapping
Delete source u-shard and ACL
Set destination u-shard ACL to R/W
Atomically:
  a. release lock on u-shard
  b. remove migration from ongoing migr. list
```

Listing 2: µ-shard migration for ZippyDB using ACLs and transactions. Writes are blocked during the migration.

If multiple replica set collections have the same highest score, we take this set of replica set collections and generate, for each, another score using resource usage data. A per-datacenter score is again generated first, which is proportional to the amount of available resources in the datacenter, taking into account, for example, CPU utilization, storage space usage, and IOPS. The per-replica set collection score is then generated by summing the individual datacenter scores on which the replica set collection has a presence. The replica set collection with the highest score is then selected for placing the target µ-shard, or a random one in case of a tie.

Information on which replica set collections are available is obtained from Configurator [35], a Facebook configuration service that each client application service keeps up to date. Listing 1 shows a simplified Akkio configuration for a sample application service. `Replication_configurations` provides a mapping between location handles and lists of datacenters in which the shard replicas are located. While location handles are opaque to Akkio, it does understand the list of datacenters and uses that information when deciding where to place µ-shards. `Consistency_requirements` specifies that this application service requires strong consistency. `Access_counter_service` specifies which data to use for the access counters. `Migration_policy` specifies a limit on the number of migrations for each µ-shard to once every 6 hours. Migrations may be limited to prevent µ-shard migration ping-ponging.

4.6.3 µ-shard migration

Once the DPS has identified a destination replica set collection for a given µ-shard, it migrates the µ-shard from the source to a destination. Different µ-shard migration methods are used, depending on the functionality of the application service's underlying database. We first describe µ-shard migration for ZippyDB, which offers access control lists (ACL's) and transactions. Other databases are considered further below. In these descriptions, we assume strong consistency of µ-shard data. We also assume the systems run reliably during migration; migration failure handling is described in (§4.6.5).

[6] Policies can be configured to include specific thresholds that shouldn't be breached; e.g. to not consider datacenters with over $n\%$ CPU usage.

```
Atomically:
  a. acquire lock on u-shard
  b. add migration to ongoing migrations list
Start double-writing to src & dest
Wait for location info cache TTL to expire
Copy data from source to dest
Switch reading to dest
Wait for location info cache TTL to expire
Switch writing to dest (ending dbl-writes)
Wait for location info cache TTL to expire
Remove src
Atomically:
  a. release lock on u-shard
  b. remove migration from ongoing migr. list
```

Listing 3: μ-shard migration for Cassandra using timestamps and double-writes. Writes are not blocked during the migration. The timestamps are used to merge data when copying

Listing 2 lists the method we first used for ZippyDB. First, a lock is acquired on the μ-shard to prevent other DPS instances from migrating the same μ-shard. (The lock does not prevent the client from reading and writing μ-shard data.) The source μ-shard ACL is then set to read only (R/O). This effectively blocks writes for the duration of the migration; however, the ZippyDB client library embedded in the application will automatically retry the write if the previous attempt was blocked, thus hiding blocked writes from the client application service.[7] The source μ-shard is then read and subsequently written to the destination μ-shard and the destination μ-shard ACL is set to R/O. The location database is updated with the new μ-shard mapping. The source μ-shard and its ACL is deleted, the destination μ-shard ACL is set to R/W, and the migration lock is released.

Not all underlying databases support ACLs. For example, the variant of Cassandra currently used at Facebook does not offer ACLs. Hence, a different migration method is needed. (See Listing 3.) In this case, the migration method takes advantage of the fact that Cassandra offers timestamps natively and can thus allow writes during ongoing migrations. After first acquiring a lock on the μ-shard, the location database information associated with the μ-shard is modified so that client writes are double written to both the source and destination, while reads continue to be directed to the source. The μ-shard data (from before the start of the double-writing) is then copied from the source to the destination. The timestamps associated with each write are used to merge data appropriately. Once the copy is complete, the location database is modified to have reads go to the destination, while continuing double-writing. The location database is modified to have writes only go to the destination. Finally, the data at the source can be deleted at the source, the μ-shard lock can be released, and the migration can be removed from the list of ongoing migrations.

With this method, each time the location database is updated, which occurs three times, it is necessary to wait for the location database TTL to expire to ensure no stale accesses go to the wrong destination. This delay could be avoided if the underlying database supports ACLs (as, e.g., open source Cassandra does), or if cache entries could be reliably invalidated, then the wait times could be reduced substantially. Also note that a potential race condition could occur with double-writes: if a write on the source succeeds, but not on the destination, then the write is observable when reading from the source, but not when later reading from the destination. We address this by always first writing to the destination, before writing to the source, on double writes.

4.6.4 Replica set collection changes

The replica set collections available to the client application service, and in particular the set of replication topologies they represent, will change over time; e.g., to account for shifts in request traffic or because of changes in underlying hardware availability. Adding a new replica set collection is straightforward: it is simply added to the configuration state and the DPS can begin to use it, migrating μ-shards to it when beneficial. Removing a replica set collection is, however, more involved. The replica set collection to be removed is first *disabled* in the configuration, preventing the DPS from selecting this shard replica set collection from future placement decisions. Then, in an off-line process, a DPS `evaluatePlacement()` call is made for each μ-shard in the disabled shard, which will cause the DPS to migrate the μ-shard to another shard replica set collection using the processes described above.

4.6.5 DPS fault recovery

When any of the servers running Akkio's location or counter services ceases to execute (say, due to a hardware or software failure), they can simply be restarted since their data is reliably persisted. The situation is different with a DPS server, since it may have been in the middle of migrating μ-shards.

To deal with this case, every DPS server instance is assigned a monotonically increasing sequence number (which is obtained from a global Zookeeper deployment [19]). This sequence number is persisted with all state related to pending migrations; e.g., in the per μ-shard lock that is acquired prior to beginning of a migration. When a DPS server instance fails, it will be restarted, potentially on a different server, with a higher sequence number. The newly restarted DPS instance will then go through a recovery process where it queries the

[7] With our ViewState workload, which has a very low read-write ratio, writes are retried in 0.007% of all accesses.

Database	Changes to datastore client library	Datastore-specific migration logic
ZippyDB C++	100	1,000
ZippyDB PHP	150	
Cassandra	500	700
Queue datastore	100	250
Datastore-X	100	250

Table 3: Lines of code implementing the two touch points between Akkio and underlying databases.

location database to identify any ongoing migrations that were initiated by the failed DPS server instance but did not complete. The sequence number for any recovered migration is updated in order to avoid any conflicts with a stale, failed DPS server instance.

For each recovered migration, the DPS servers identifies which state to continue the migration on. This is a custom piece of code that is different for each underlying datastore system and migration approach used. For example, in our ACL based approach, the DPS scans the state of the µ-shard in the source backend and the destination backend to identify which steps of the migration had been completed. It then resumes the migration from that point on. In case of errors during a single migration step, we restart the migration. Migrations are typically retried until they succeed (although this is configurable).

5 Evaluation

5.1 Implementation metrics

A benefit of Akkio's design that enhances portability is how lightweight the touchpoints are between Akkio and the underlying databases. Table 3 lists the lines of code (LoC) required for each of the two touchpoints: e.g., the ZippyDB client library only required 100-150 new or modified LoC to accommodate Akkio, and Akkio only required 1,000 or fewer LoC of datastore-specific code for µ-shard migrations in ZippyDB.

5.2 Use cases analysis

We describe the effect Akkio had on 4 different client application services. All of the metrics we present were gathered from our live production systems running at scale, driven by live user traffic. This limits our ability to experiment, so we primarily compare against the systems that were in place before Akkio was introduced.

5.2.1 ViewState

Description: ViewState stores a history of content previously shown to a user. Each time a user is shown some content, an additional snapshot is appended to the ViewState data. The data is used to prioritize subsequent content each time it is displayed to the user. ViewState stores this history, with an average size of 500KB, in ZippyDB.

Requirements: ViewState data is read on the critical path when displaying content, so minimizing read latencies is important. Writes are not on the critical path, but low write latencies are important for the application, as user engagement tends to drops if the content is not "fresh". The data needs to be replicated three ways for durability. Strong consistency is a requirement.

Setup: ViewState uses replica set collections configured with two replicas in one (local) datacenter and a third in a nearby datacenter with the primary preferentially located in the local datacenter. Akkio migrates µ-shards aggressively for ViewState. Having the primary replica be local ensures reads are fast. Having two replicas locally ensures writes are fast given that a quorum exists locally. Having two replicas locally has the further advantage that, should the primary fail, then the other can become primary. In aggregate, 6 different replica set collections are available for Akkio to migrate ViewState µ-shards across when using 6 datacenters.

Having the primary plus a replica in the same datacenter could, however, cause some writes to get lost should an entire datacenter go down: writes that have reached the primary and the other replica in the same datacenter, but have not reached the third replica, will get lost. The ViewState owners were willing to make this tradeoff for this rare scenario.

Result: Originally, ViewState data was fully replicated across six datacenters. Using Akkio with the setup described above led to a 40% smaller storage footprint,[8] a 50% reduction of cross-datacenter traffic, and about a 60% reduction in read and write latencies compared to the original non-Akkio setup. Each remote access notifies the DPS, resulting in approximately 20,000 migrations a second. See Fig. 7. Using Akkio, roughly 5% of the ViewState reads and writes go to a remote datacenter.

5.2.2 AccessState

Description: AccessState stores information with respect to user actions taken in response to content displayed to the user. The information includes the action taken, what content it was related to, a timestamp of when the action was taken, and so on. AccessState data is appended to by a number of different services, but read mostly by the dynamic content display system. AccessState stores the action history, with an average size of 200KB, in ZippyDB. The read-write ratio for AccessState is far lower than it is for ViewState.

Requirements: Reads are on the critical path when deciding what content to display, and hence low read la-

[8] Only 40% because the number of servers couldn't be further reduced due to the CPU becoming the bottleneck.

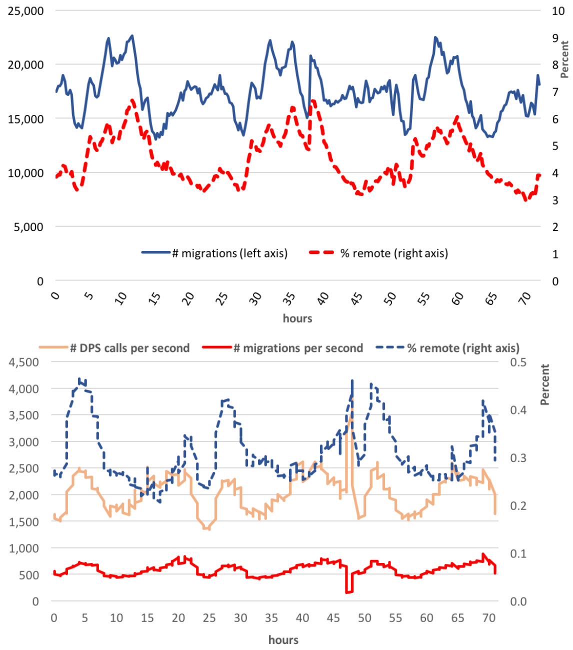

Figure 7: ViewState (top); AccessState (bottom): percentage of accesses to remote data, the number of `evaluatePlacement()` calls to DPS per second, and the number of ensuing µ-shard migrations per second. For ViewState the number of calls to DPS per second and the number of migrations per second are the same.

	avg	p90	p95	p99
With Akkio:	10ms	23ms	26ms	34ms
Without Akkio	76ms	151ms	237ms	371ms

Table 4: AccessState client service access latencies.

tencies are needed. However, writes are not on the critical path and moderate write latency is acceptable (unlike ViewState). The data needs to be replicated three ways but only needs to be eventually consistent.

Setup: AccessState uses replica set collections configured to have three replicas, each one in a different datacenter. Overall, 20 such replica set collections, each with a different topology configuration, plus one replica set collection configured to have a replica in each datacenter, are available for Akkio to migrate AccessState µ-shards. Akkio is configured to not migrate µ-shards aggressively if, based on the access history, it believes the remote processing may be transient. Moreover, it does not migrate the primary replica to the datacenter from which the access was made even though it would lead to lower write latencies, mainly because not doing so significantly reduces the number of migrations needed. (Note that the read-write ratio for AccessState is far higher than it is for ViewState.)

Result: Originally, AccessState data was configured to be fully replicated across six datacenters. Using Akkio with the setup described above led to a 40% decrease in storage footprint, a roughly 50% reduction of cross-datacenter traffic, negligible increase in read latency (0.4%) and a 60% reduction in write latency. Roughly 0.4% of the reads go remote, resulting in about 1,000 migrations a second. Figure 7 shows that there are roughly half as many migrations as there are calls to the DPS.

We also compared AccessState read latencies for a configuration with 3X replication, with and without Akkio. For the configuration without Akkio, the replicas were spread evenly across all datacenters. The results are shown in Table. 4: without Akkio, access latencies are 7X–10X higher.

5.2.3 Instagram Connection-Info

Description: Connection-Info stores data for each user, including when and from where they were online, as well as other status and connection endpoint information. This data is stored on Cassandra. There are roughly 30 billion µ-shards.

Requirements: This application service requires strong consistency, for which it uses Cassandra's quorum read and write features [18]. Intra-continental quorum read and write latencies are important. Originally, this service stored its data using full replication across five datacenters on one continent, but as usage in a second continent increased substantially, some of the data had to be stored on that continent.

Setup: This service uses two replica configurations. One has 5X replication, with a replica in each of five datacenters (as its original setup). The second has 3X replication with two in the second continent and one in the first. Having two replicas together ensures a quorum stays within the same continent in the steady state.

Result: With Akkio it was possible to keep both read and write latencies lower than 50ms which was important to its operation, compared to greater than 100ms which would have been incurred if quorums went across datacenters. This service could not have expanded into the second continent without Akkio.

5.2.4 Instagram Direct

Description: This is a traditional messaging application service that supports group messaging. Each message queue contains the sent messages as well as "cursors" that track the position in the queue for each subscriber. There are roughly 15 billion such queues, but with most queues having a small footprint of a few hundred bytes. The messaging application relies on Iris, a specialized Facebook-internal queuing datastore service that guarantees in-order delivery. (Underneath, Iris uses MySQL for persistent storage.)

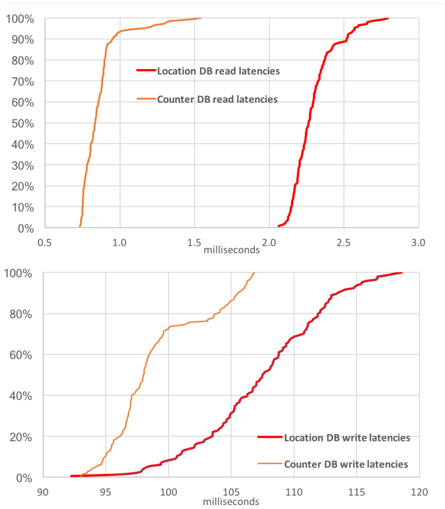

Figure 8: Distribution of client-side latencies for accessing the Akkio location and counter databases, (not taking the cache into account). Read latencies are shown in the top graph; write latencies in the bottom graph.

Requirements: Iris is on the critical path for Instagram Direct end-to-end message delivery. Both low write and low read latencies are thus important. Strong consistency is required.

Setup: Currently, three datacenters are used to store Instagram Direct data. The database is configured to have replica configurations with a primary in each datacenter. Further, each replica set has a secondary replica in the same datacenter as the primary and two additional replica in another datacenter, for a total of four replicas. User access history information is used to decide where to place µ-shards; for message queues that are accessed by multiple users (i.e., group messaging) placement is determined by using each user's access history weighted by rate of user actions.

Result: With Akkio, on average, roughly 3,000 migrations occur per second, resulting in a reduction in end-to-end message delivery latency by 90ms at p90 and 150ms at p95. This, in turn, resulted in user engagement improvements, where the number of message sends increased by 0.9% overall and the number of text message sends increased by 1.1%.

5.3 Analysis of Akkio services

Location Service Using AccessState as an example, the location database uses roughly 200GB storage space (unreplicated) to keep track of the location of each µ-shard, with one µ-shard for each of Facebook's billion+ users. The location database is itself one of the use cases that shares a multi-tenant ZippyDB deployment. It consumes 1,200 fully replicated shards with the primary replicas spread evenly across all regions.

The hit rate of the distributed front-end cache is 98% on average. Read latency on the cache averages to around 1 ms. Figure 8 show the distributions of Akkio client library-side read and write latencies after a miss in the cache. Writes take considerably longer because a quorum needs to be achieved across datacenters before a write is acknowledged.

Step	Time (avg.)
Acquire Lock	151ms
Set Source ACL To Read Only	315ms
Read µ-shard from Source	184ms
Write µ-shard to Destination	130ms
Update Location in DB	151ms
Delete µ-shard From Source	160ms
Set Destination ACL to Read Write	120ms
Release Lock	151ms

Table 5: Breakdown for AccessState µ-shard migration times.

Access Counter Service We present various metrics from the Access Counter DB for AccessState as an example. The amount of storage required for storing one counter for each of the billion+ users and datacenter is about 400GB in total (unreplicated). The Access Counter database also lives in our ZippyDB multi-tenant deployment with 1,100 dedicated shards. Figure 8 depicts the counter database read and write latencies. Neither the reads nor the writes on this database are on any critical path. The read-write ratio is about 1:500. In a typical day, the Access Counter DB for ViewState processes between 300,000 and 550,000 writes per second.

Data Placement Service The DPS receives about 100,000 `evaluatePlacement()` calls per second. However, these calls are asynchronous and not on any critical path. Migrations are the heavy-weight operations executed by the DPS. Table 5 shows the elapsed time breakdown of an AccessState µ-shard migration. The sum of all the individual latencies is relatively high; however, some of the operations can be executed in parallel, different migrations can proceed in parallel, and migration itself is not on the critical path. These latencies have not been an issue for the client application services using Akkio today; optimizing them is left for future work.

6 Related Work

Almost all datastore systems have some form of sharding in order to be scalable, and offer replication to provide high availability; e.g., [6, 10, 16, 30, 33]. However, these systems offer little in terms of locality management. For example, while Cassandra supports fine-grained control of cross-datacenter replication, the control is static and not based on access patterns [23].

A number of systems manage data locality at shard granularity [4, 12, 29, 40]. Given their typical size, we argue that it is challenging to place shards so that most data accesses are local if the number of replicas is limited. Moreover, the overhead of migrating entire shards is high, and hence these systems tend to be slow to react to shifts in workload.

A few systems manage data locality at a granularity finer than shards. Spanner supports µ-shards in the form of *directories* [11], its unit of data placement. Applications control the contents of a directory using commonality in key prefixes. However, [11] makes no mention of directory-level locality management.

Kadambi et al. extend Yahoo! PNUTs [10] with a per-record selective replication policy [20] but only offer eventual consistency. PNUTs behaves similarly to a distributed cache in that some replicas of records are transient and created on reads and removed when stale; however data resides on disk and a (configurable) minimum number of replicas are kept up to date by propagating updates. The authors argue that collecting and maintaining access statistics of individual records is too complex and incurs too much overhead. Akkio's design shows this need not be. Not tracking these fine grained statistics can lead to sub-optimal decisions.

Volley determines where to place data based on logs that must capture each access [1]. It does this at object granularity. It generates placement and migration recommendations, but leaves the coordination and execution of any resulting migrations to the application, thus making it cumbersome for an application to integrate it. Volley's design to process access logs offline makes it slow to react to shifts in workload and to other real-time events.

Nomad is a prototype distributed key-value store that supports *overlays* as an abstraction [36] designed to hide the protocols needed to coordinate access to data as it is migrated across datacenters. The unit of data management is a *container*, which corresponds to an Akkio µ-shard. However, Nomad does not track access histories or take capacities, loads, and resource-effectiveness into account as Akkio does.

7 Concluding Remarks

This paper makes two key contributions. First, we introduce Akkio, a dynamic locality management service. Second, we introduce and advocate for a finer-grained notion of datasets called µ-shards. To our knowledge, Akkio is the first dynamic data locality system for geo-distributed datastore systems that migrates data at µ-shard granularity, that can offer strong consistency, and that can operate at scale. The system demonstrates that it is possible, and advantageous, to capture data access statistics at fine granularity for making data placement decisions.

Akkio's design is reasonably simple and largely based on techniques well-established in the distributed systems community. Yet we have found it to be effective (§5.2). So far, several hundred application services at Facebook use Akkio, and Akkio manages over 100PB of data. We believe that our choice to implement Akkio as a separate layer between the application services and their underlying databases has worked out well. Separating the concerns of locality management on the one hand, and replication, load-balancing and failure recovery on the other hand, led to a much simpler design and made Akkio viable to a larger set of application services.

With our experiences deploying Akkio, we learned a number of lessons, most of which center around having to make Akkio far more configurable than we had anticipated. (1) we initially planned on storing all of Akkio's metadata in Akkio's own datastore system (ZippyDB). However, we found that application service owners were not willing to add an extra cross-datastore dependency in their critical path (and not willing to change the underlying datastore system they were already using). This forced us to make the location metadata store logic pluggable so that the location metadata could be stored on the application's underlying datastore system. (2) We initially assumed all application services would follow the same migration strategy. However, we found that we had to create a separate migration strategy for each underlying datastore system so as to play to its strengths. (3) We learned that migrations didn't need to be real-time in all cases; e.g., moving messenger conversations to their center of gravity once a day lead to more efficient resource usage, in part because smarter, off-line placement decisions became feasible. More generally, we found that the decision of when to migrate had to be customizable: many application services wanted to delay having their µ-shards migrated by several hundred milliseconds after the first sub-optimal access in order to decrease the chances of the migration interfering with subsequent write accesses (especially if the migration strategy involved taking the µ-shard offline to writes for a small duration). (4) We expected to only need a few different scoring policies when making placement decisions, but ultimately had to support quite a variety of specific scoring policies; e.g., taking recent activity of individual end-users into account when making messaging µ-shard placement decisions. (5) We found that Akkio made capacity planning (growth projections for different datacenters) significantly more difficult with the added dimension of locality, requiring finer-grained estimates of datacenter resource growth.

Going forward, more applications are being moved to run on Akkio, and more datastore systems are being supported (e.g., MySQL). Further, work has started using Akkio (*i*) to migrate data between hot and cold storage, and (*ii*) to migrate data more gracefully onto newly created shards when resharding is required to accommodate (many) new nodes.

Acknowledgements

Many helped contribute to the Akkio system; in particular Victoria Dudin, Harsh Poddar, Dmitry Guyvoronsky; from the ZippyDB team: Sanketh Indarapu, Sumeet Ungratwar, Benjamin Renard, Daniel Pereira, Prateek Jain, Renato Ferreira, Joanna Bujnowska, Igor Pozgaj, Charlie Pisuraj, Tim Mulhern; from the Cassandra team: Dikang Gu, Andrew Whang, Xiangzhou Xia, Abhishek Maloo; from the Generic Iris team: Changle Wang, Jeremy Fein, Kristina Shia; From the Instagram team: Colin Chang, Jingsong Wang; from the Messaging Iris team: Rafal Szymanski, Jeffrey Bahr, Phil Lopreiato, Adrian Wang. We also thank the reviewers, and our shepherd Kang Chen, for their constructive comments that led to a far better paper.

References

[1] Agarwal, S., Dunagan, J., Jain, N., Saroiu, S., Wolman, A., and Bhogan, H. Volley: Automated data placement for geo-distributed cloud services. In *Proc. 7th USENIX Conf. on Networked Systems Design and Implementation (NSDI'10)* (San Jose, California, April 2010), pp. 17–32.

[2] Amiri, K., Park, S., Tewari, R., and Padmanabhan, S. DBProxy: A dynamic data cache for web applications. In *Proc. 19th Intl. Conf. on Data Engineering (ICDE'03)* (Bangalore, India, March 2003), pp. 821–831.

[3] Annamalai, M. ZippyDB: A distributed key-value store. Talk at Data @ Scale: `https://code.facebook.com/posts/371721473024046/inside-data-scale-2015`, June 2015.

[4] Ardekani, M. S., and Terry, D. B. A self-configurable geo-replicated cloud storage system. In *Proc 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, October 2014), pp. 367–381.

[5] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H. C., et al. TAO: Facebook's distributed data store for the social graph. In *Proc. USENIX Annual Technical Conference (USENIXATC'13)* (San Jose, CA, June 2013), pp. 49–60.

[6] Cattell, R. Scalable SQL and NoSQL data stores. *SIGMOD Rec. 39*, 4 (May 2011), 12–27.

[7] Chabchoub, Y., and Hebrail, G. Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window. In *Proc. IEEE Intl. Conf. on Data Mining Workshops* (Sydney, Australia, Dec 2010), pp. 1297–1303.

[8] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S. Realtime data processing at Facebook. In *Proc. 2016 Intl. Conf. on Management of Data (SIGMOD'16)* (San Francisco, California, 2016), pp. 1087–1098.

[9] Chester, D. Considering the real cost of public cloud storage vs. on-premises object storage, June 2017. [Online; posted 23-June-2017].

[10] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. PNUTS: Yahoo!'s hosted data serving platform. *Proc. of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[11] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, Oct 2012), pp. 261–264.

[12] Curino, C., Jones, E., Zhang, Y., and Madden, S. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endowment 3*, 1-2 (Sept. 2010), 48–57.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symp. on Operating Systems Principles (SOSP'07)* (Stevenson, Washington, 2007), pp. 205–220.

[14] Fitzpatrick, B. Distributed caching with Memcached. *Linux Journal 2004*, 124 (2004), 5.

[15] Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C., and Tomasic, A. Scalable query result caching for web applications. *Proc. VLDB Endow.* (Aug. 2008), 550–561.

[16] George, L. *HBase: The Definitive Guide*, 2nd ed. O'Reilly Media, Inc., 2017.

[17] Google. Cloud locations. `https://cloud.google.com/about/locations/`. [Online; retrieved 12-April-2018].

[18] Hewitt, E., and Carpenter, J. *Cassandra: The Definitive Guide*, 2 ed. O'Reilly Media, 2016.

[19] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX Annual Technical Conference (USENIXATC'10)* (Boston, MA, 2010), pp. 145–158.

[20] Kadambi, S., Chen, J., Cooper, B. F., Lomax, D., Ramakrishnan, R., Silberstein, A., Tam, E., and Garcia-Molina, H. Where in the world is my data. In *Proc. 34th Intl. Conf. on Very Large Data Bases (VLDB'11)* (Seattle, Washington, August 2011), pp. 1040–1050.

[21] Kirsch, J., and Amir, Y. Paxos for system builders: An overview. In *Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)* (Yorktown Heights, NY, 2008), ACM, pp. 3:1–3:6.

[22] Kreifeldt, E. Myriad factors conspire to lower submarine bandwidth prices. `http://www.lightwaveonline.com/articles/2016/08/myriad-factors-conspire-to-lower-submarine-bandwidth-prices.html`, August 2016. [Online; posted 31-August-2016 — original source: TeleGeography `https://www.telegeography.com`].

[23] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[24] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[25] Lamport, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Dec 2001), 51–58.

[26] Marz, N., and Warren, J. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[27] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani,

V. Scaling Memcache at Facebook. In *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation (NSDI'13)* (Lombard, IL, 2013), pp. 385–398.

[28] NUFIRE, T. The cost of cloud storage. https://www.backblaze.com/blog/cost-of-cloud-storage, June 2017. [Online; posted 29-June-2017].

[29] P N, S., SIVAKUMAR, A., RAO, S., AND TAWARMALANI, M. D-tunes: Self tuning datastores for geo-distributed interactive applications. In *Proc. of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)* (Hong Kong, 2013), pp. 483–484.

[30] PLUGGE, E., HOWS, D., MEMBREY, P., AND HAWKINS, T. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB*, 3rd ed. Apress, 2015.

[31] ROWLING, J. K. *Harry Potter and the Goblet of Fire*. Thorndike Press, 2000.

[32] SHAROV, A., SHRAER, A., MERCHANT, A., AND STOKELY, M. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. of the VLDB Endowment 8*, 12 (2015), 1490–1501.

[33] STRICKLAND, R. *Cassandra 3.x High Availability*, 2nd ed. Packt Publishing Ltd, 2016.

[34] TAI, A., KRYCZKA, A., KANAUJIA, S., PETERSEN, C., ANTONOV, M., WALIJI, M., JAMIESON, K., FREEDMAN, M. J., AND CIDON, A. Live recovery of bit corruptions in datacenter storage systems. *CoRR abs/1805.02790* (2018).

[35] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at Facebook. In *Proc. 25th Symp. on Operating Systems Principles (SOSP'15)* (Monterey, California, 2015), pp. 328–343.

[36] TRAN, N., AGUILERA, M. K., AND BALAKRISHNAN, M. Online migration for geo-distributed storage systems. In *Proc. USENIX Annual Technical Conference (USENICATC'11)* (Portland, Oregon, June 2011), pp. 201–215.

[37] WIKIPEDIA CONTRIBUTORS. Shard (database architecture) — Wikipedia. https://en.wikipedia.org/w/index.php?title=Shard_(database_architecture)&oldid=845931919, 2018. [Online; accessed 14-September-2018].

[38] WOODS, A., AND HARRISON, D. How to leverage geo-partitioning. https://www.cockroachlabs.com/blog/geo-partitioning-two/, April 2018. [Online; retrieved 12-April-2018].

[39] WU, Z., BUTKIEWICZ, M., PERKINS, D., KATZ-BASSETT, E., AND MADHYASTHA, H. V. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. 24th ACM Symp. on Operating Systems Principles (SOSP'13)* (Farminton, Pennsylvania, November 2013), pp. 292–308.

[40] YU, H., AND VAHDAT, A. Minimal replication cost for availability. In *Proc. 21st Annual Symp. on Principles of Distributed Computing (PODC'02)* (Monterey, California, July 2002), pp. 98–107.