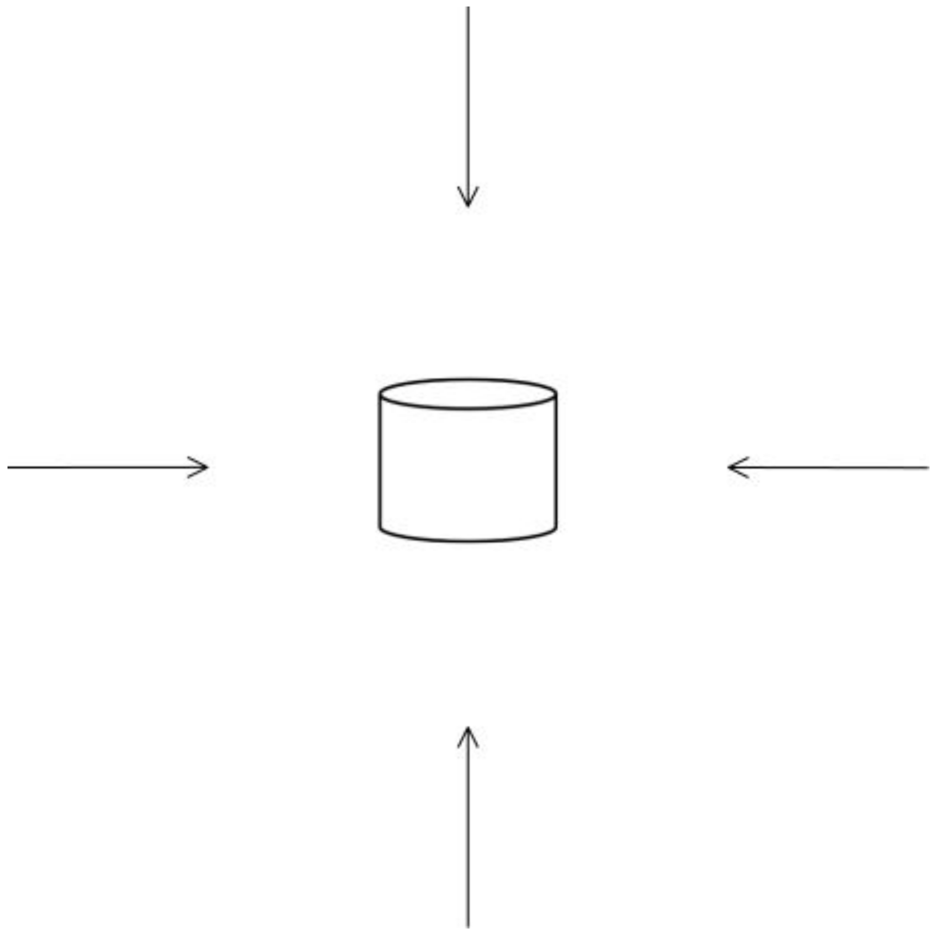
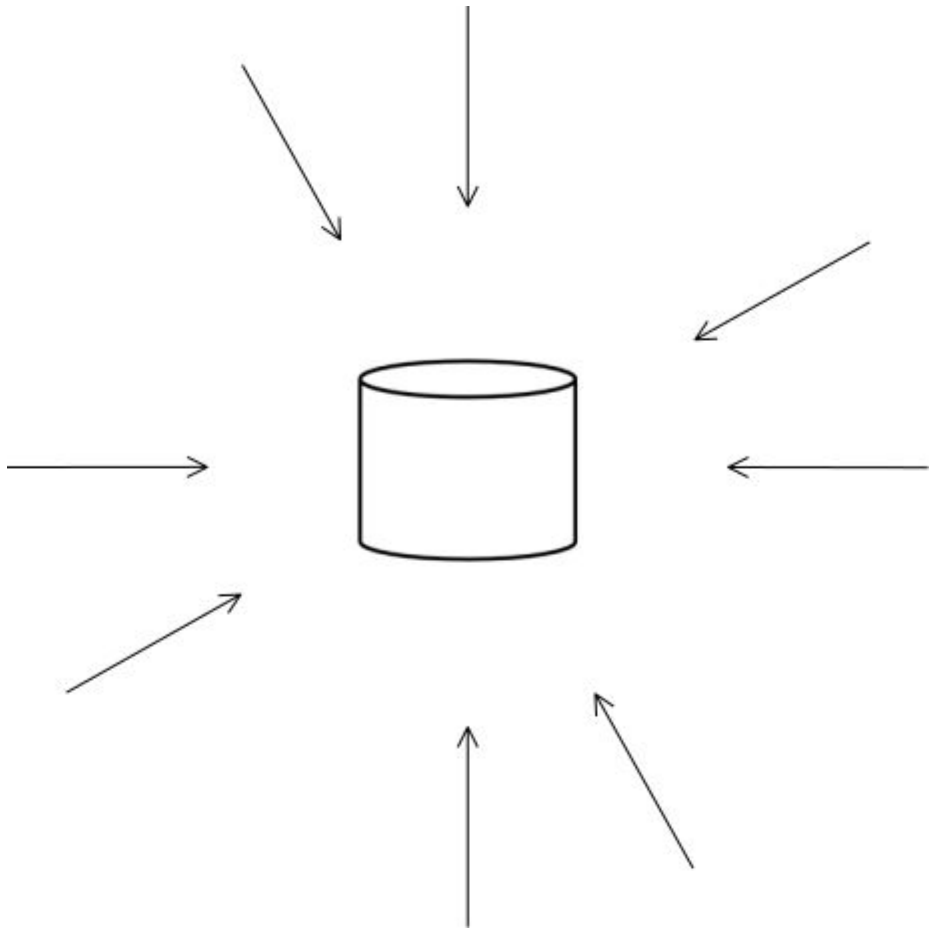
# Sharding the Shards: Managing Datastore Locality at Scale with Akkio

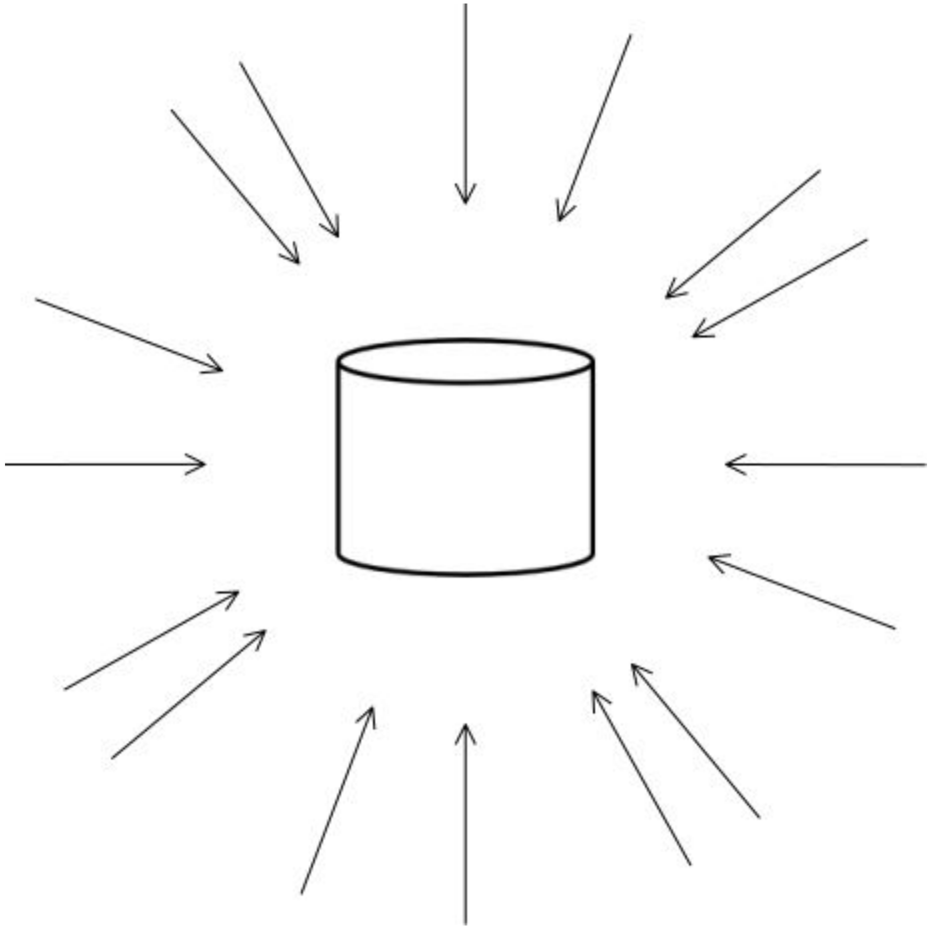
Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, and David Nagle, Facebook; Michael Stumm, University of Toronto

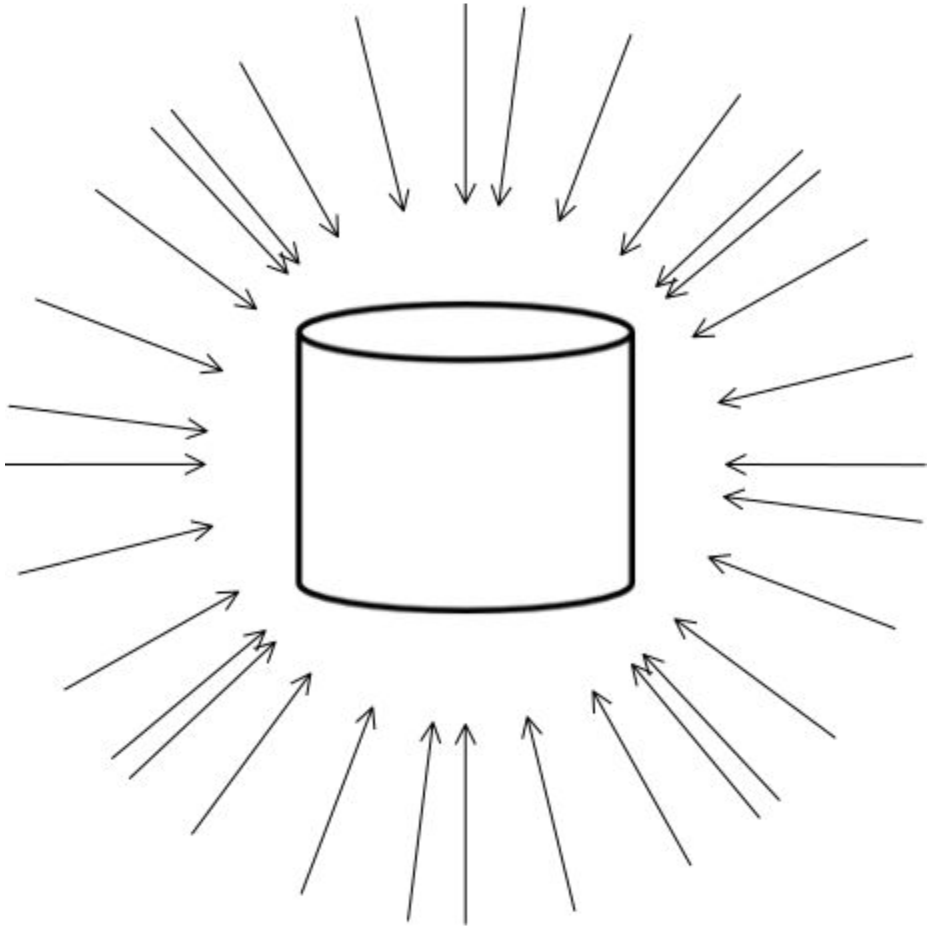
# Databases

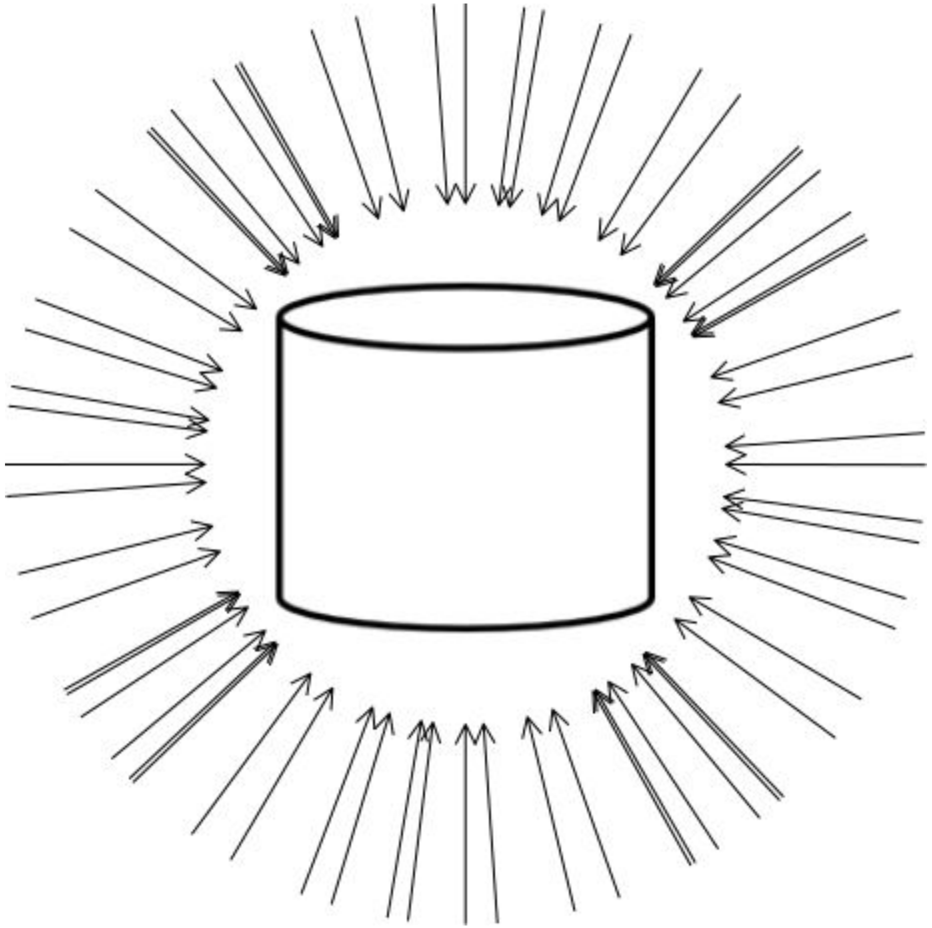
Kamil Braun, University of Warsaw

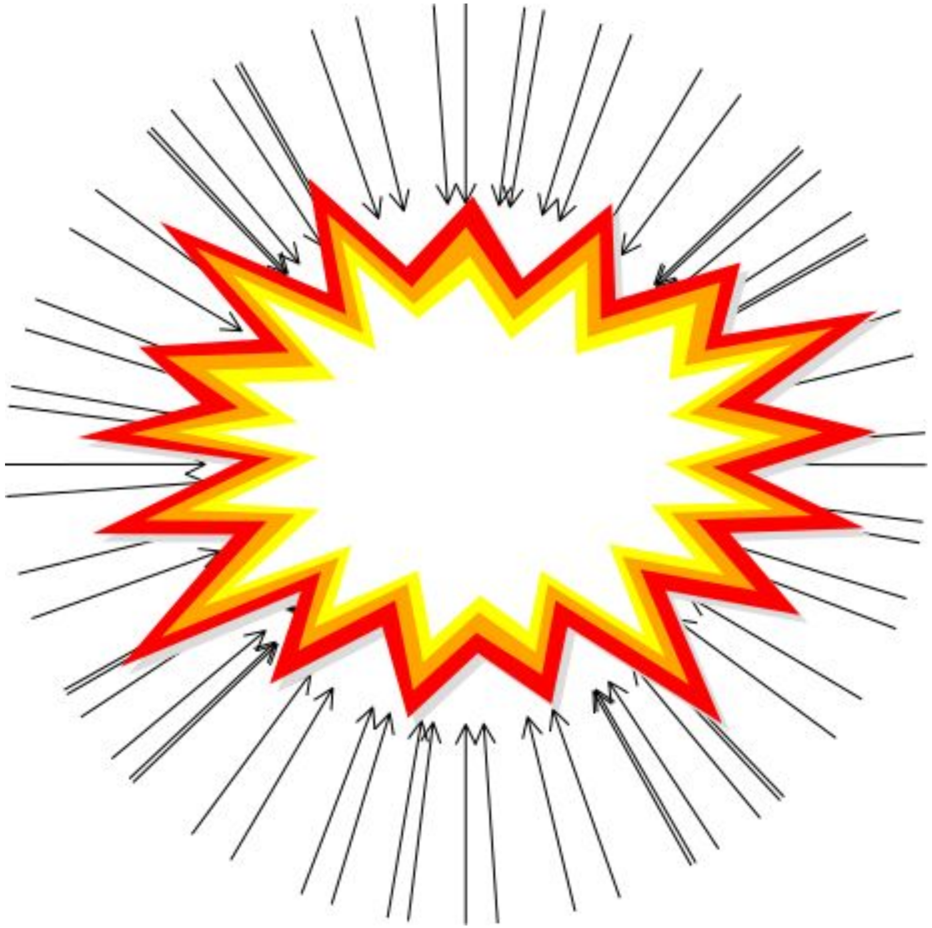












## Distributed databases

## How to make distributed databases

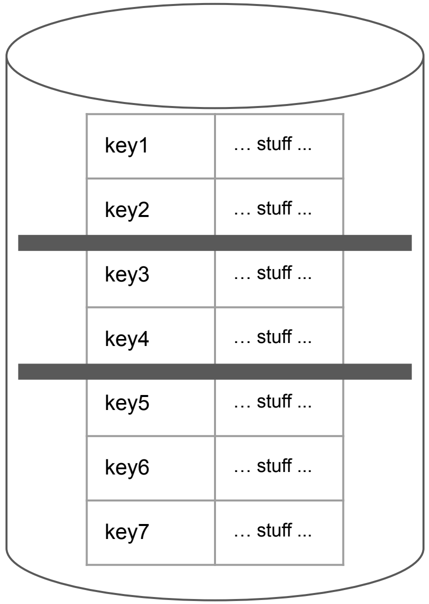
which don’t not work

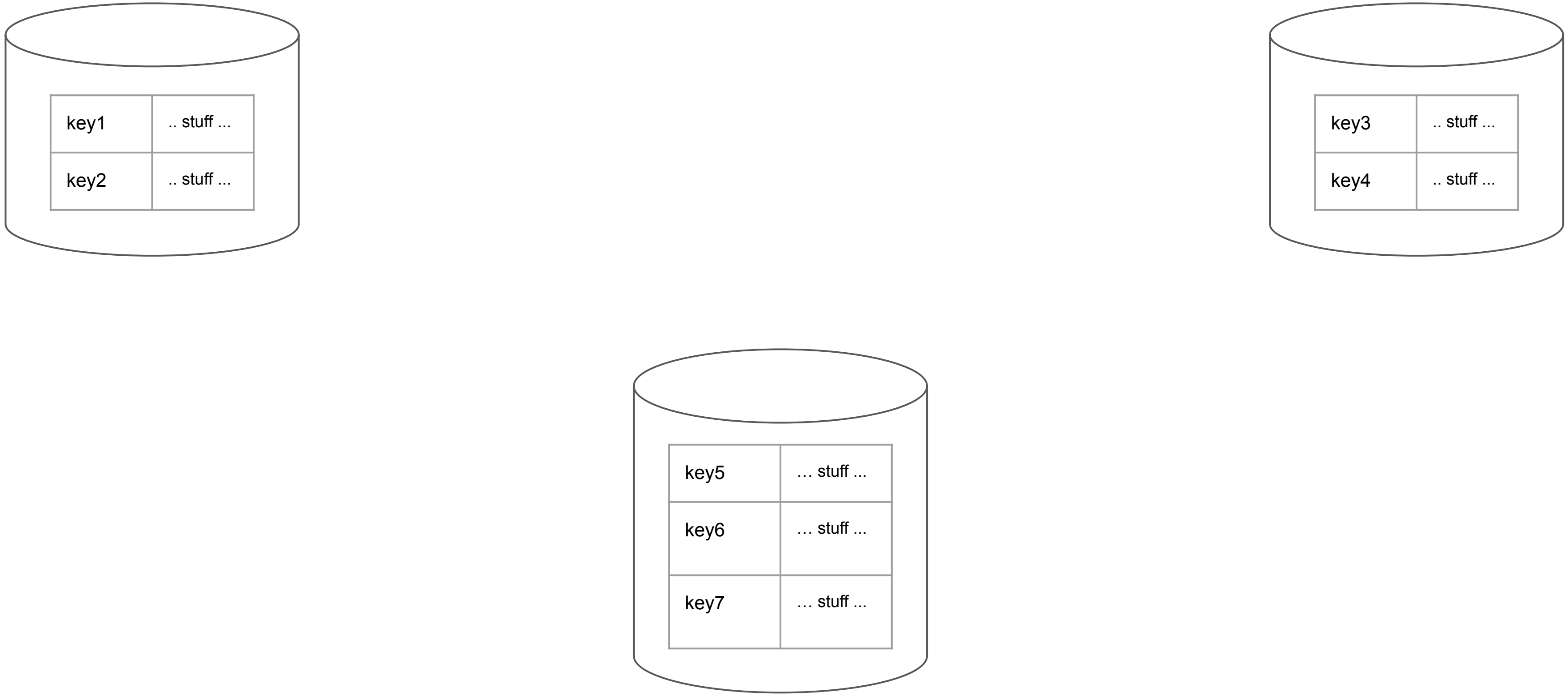
## Horizontal partitioning

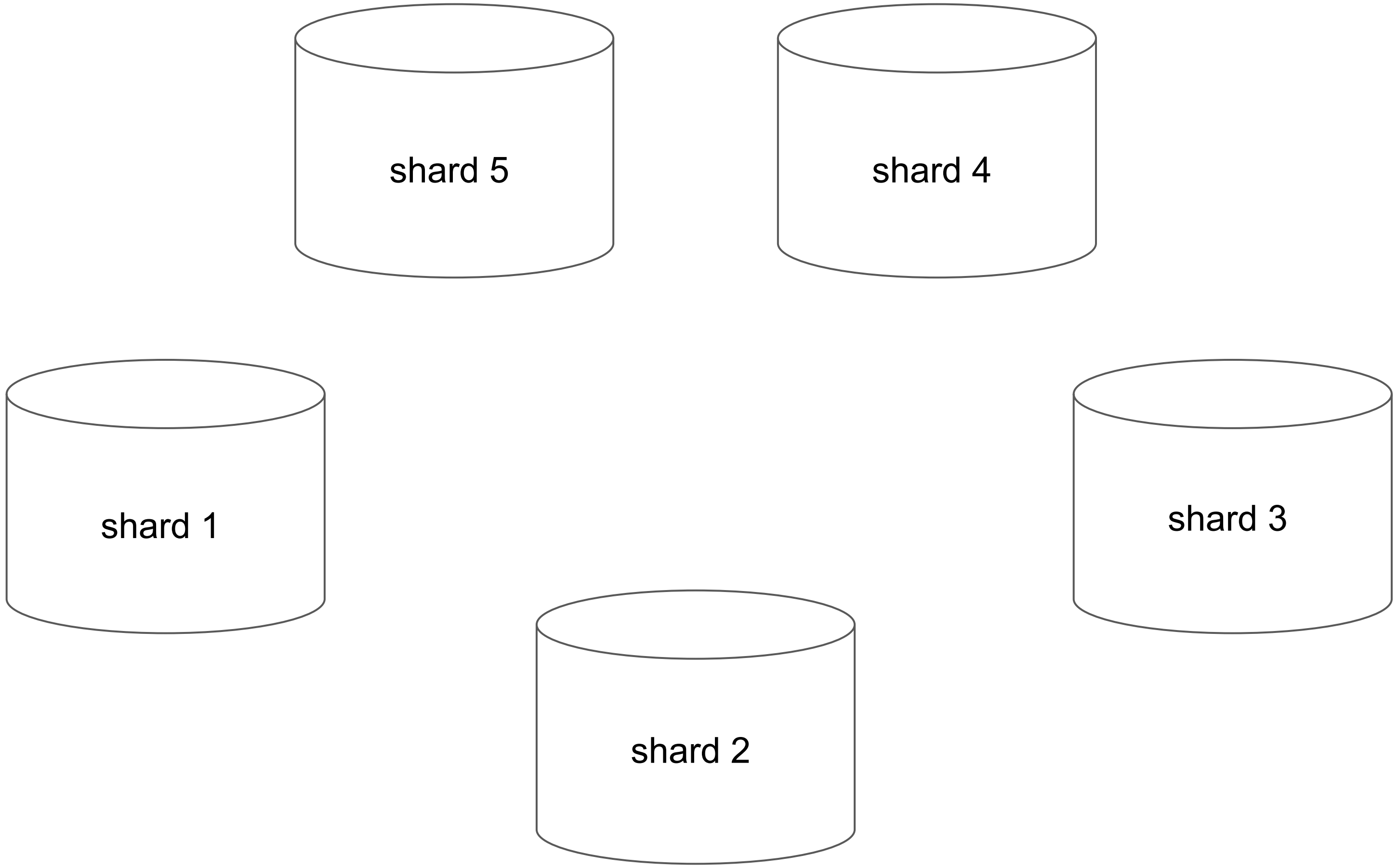
(or “sharding”)

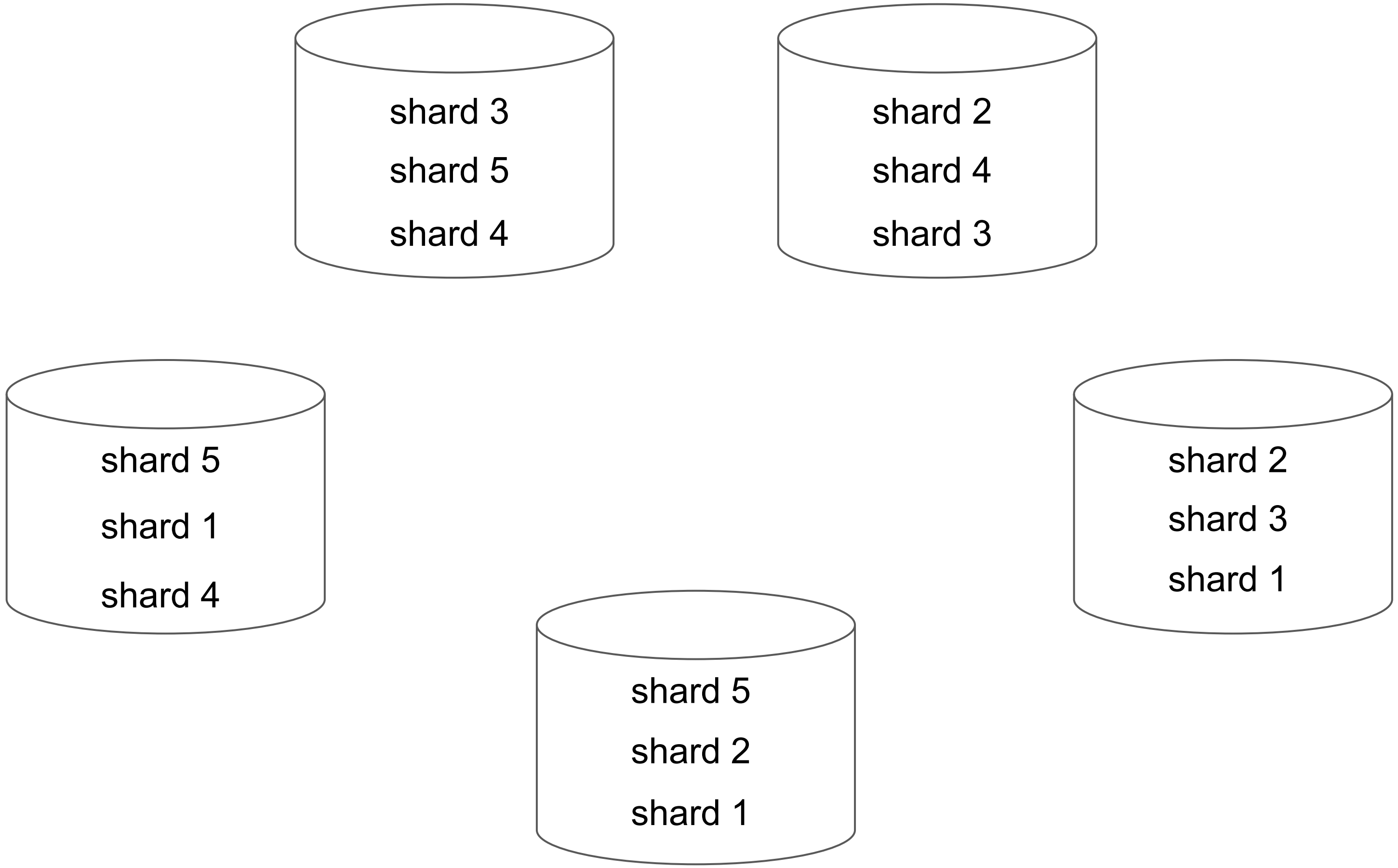
| | |
|---|---|
| key1 | … stuff ... |
| key2 | … stuff ... |
| key3 | … stuff ... |
| key4 | … stuff ... |
| key5 | … stuff ... |
| key6 | … stuff ... |
| key7 | … stuff ... |

| | |
|---|---|
| key1 | … stuff ... |
| key2 | … stuff ... |
| key3 | … stuff ... |
| key4 | … stuff ... |
| key5 | … stuff ... |
| key6 | … stuff ... |
| key7 | … stuff ... |


key1
… stuff ...
key2
… stuff ...
key3
… stuff ...
key4
… stuff ...
key5
… stuff ...
key6
… stuff ...
key7
… stuff ...


key1
.. stuff ...
key2
.. stuff ...
key3
.. stuff ...
key4
.. stuff ...
key5
… stuff ...
key6
… stuff ...
key7
… stuff ...


shard 5
shard 4
shard 1
shard 3
shard 2


shard 3
shard 5
shard 4
shard 2
shard 4
shard 3
shard 5
shard 1
shard 4
shard 2
shard 3
shard 1
shard 5
shard 2
shard 1

# ZippyDB

“a scalable key-value store”

(read: a distributed database)

## ZippyDB

- data is partitioned horizontally
- each shard is configured to have a set of replicas (a **shard replica set**)
- one is called the **primary**, the rest are **secondaries**
- writes sent to primary, which then replicates to secondaries

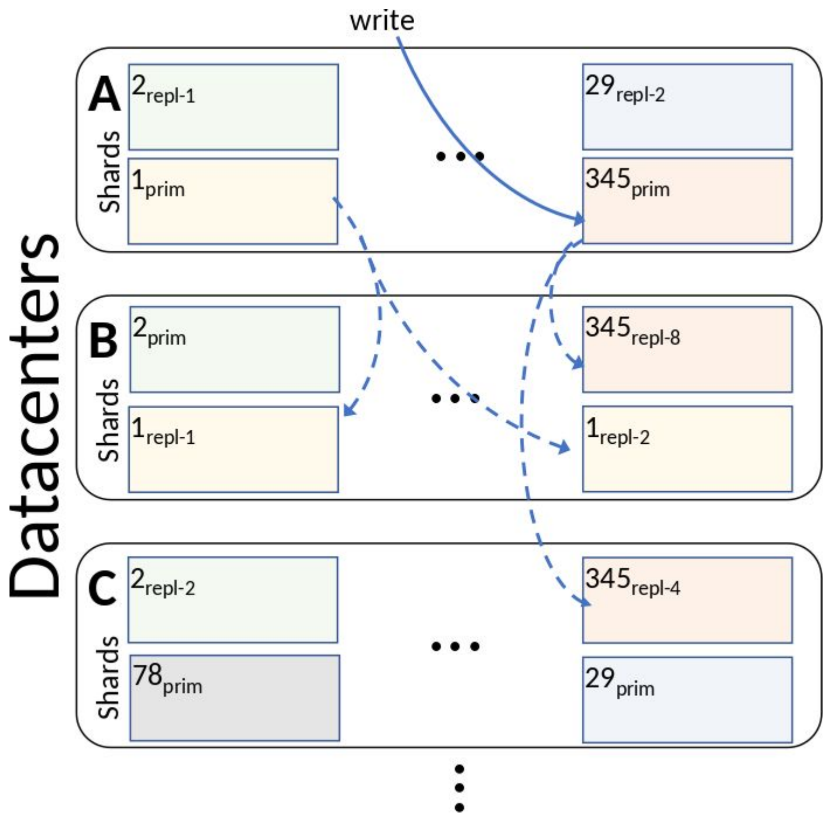
“reads that need to be strongly consistent need to be directed to the primary”

“if eventual consistency is acceptable, then reads can be directed to a secondary”

## ZippyDB

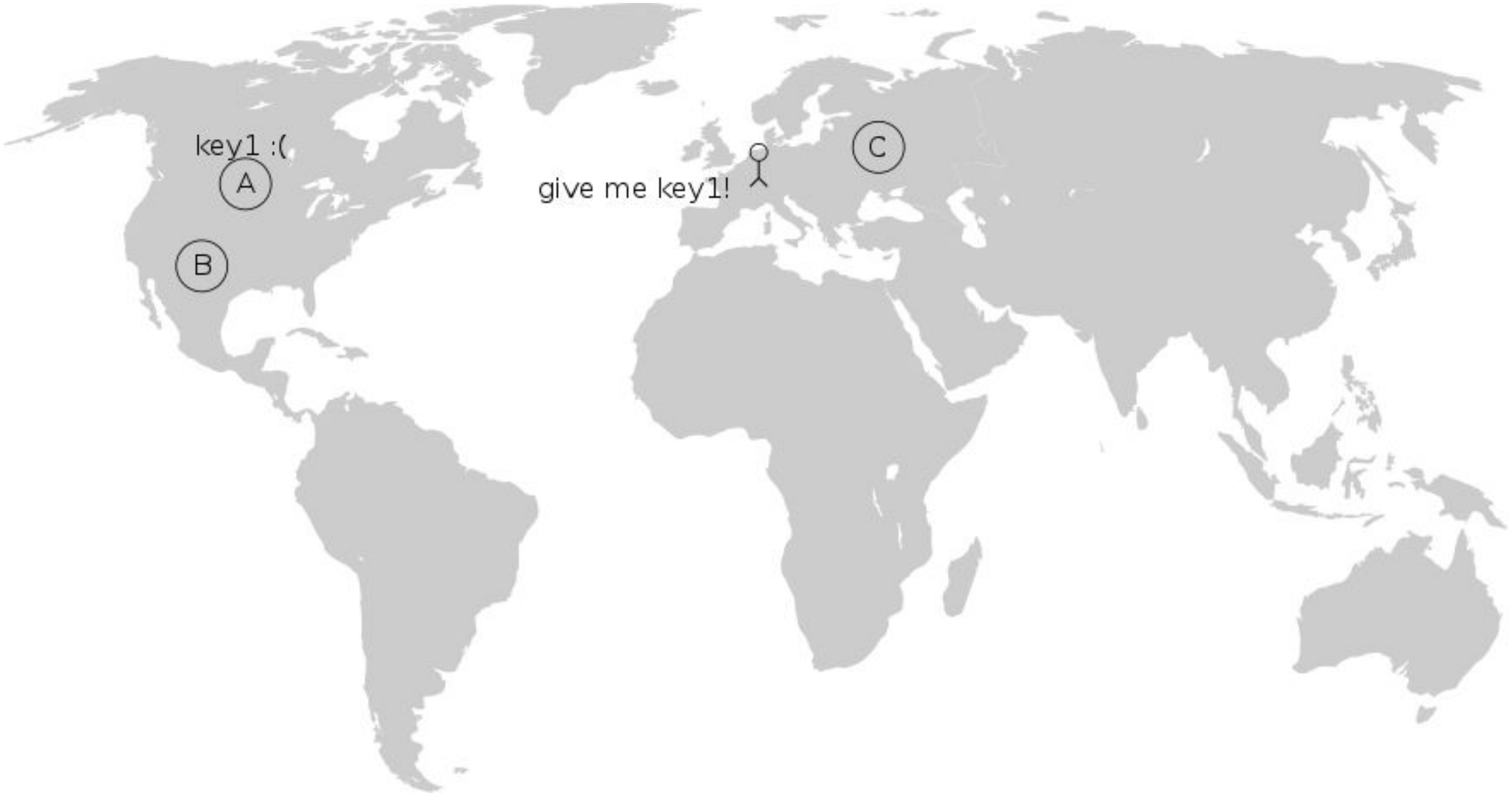
- each shard has a **replication configuration**
  i.e.: how many replicas? How are they distributed across DCs and racks?
  e.g.: “3 replicas, the primary + one secondary in one DC, the other secondary in a different DC”
- replication configuration -> **replica set collection**:
  the collection of all replica sets having this configuration
- replica set collection -> **location handle**, it’s unique ID.
- ZippyDB provisions shards and replica set collections
- user inputs parameters and constraints that they must satisfy
  e.g.: “replication factor = 3; always put 2 replicas in the same DC”
- ZippyDB assigns each shard’s replica to a physical machine
  according to the given constraints

# ZippyDB


write
Datacenters
A
Shards
$2_{repl-1}$
$1_{prim}$
$29_{repl-2}$
$345_{prim}$
B
Shards
$2_{prim}$
$1_{repl-1}$
$345_{repl-8}$
$1_{repl-2}$
C
Shards
$2_{repl-2}$
$78_{prim}$
$345_{repl-4}$
$29_{prim}$

## What shards don’t give?

## Data access locality



## Data access locality

- = low latencies, e.g.
  - intra-DC communication latencies: ~1ms
  - cross-DC latencies: ~100ms
- = low cross-DC bandwidth usage

## Properties of shards

- unit of replication
- failure recovery
- load balancing
- some arbitrary subset of keys (e.g., according to a hash)
- size: usually tens of GBs
- number: a couple of tens of thousands

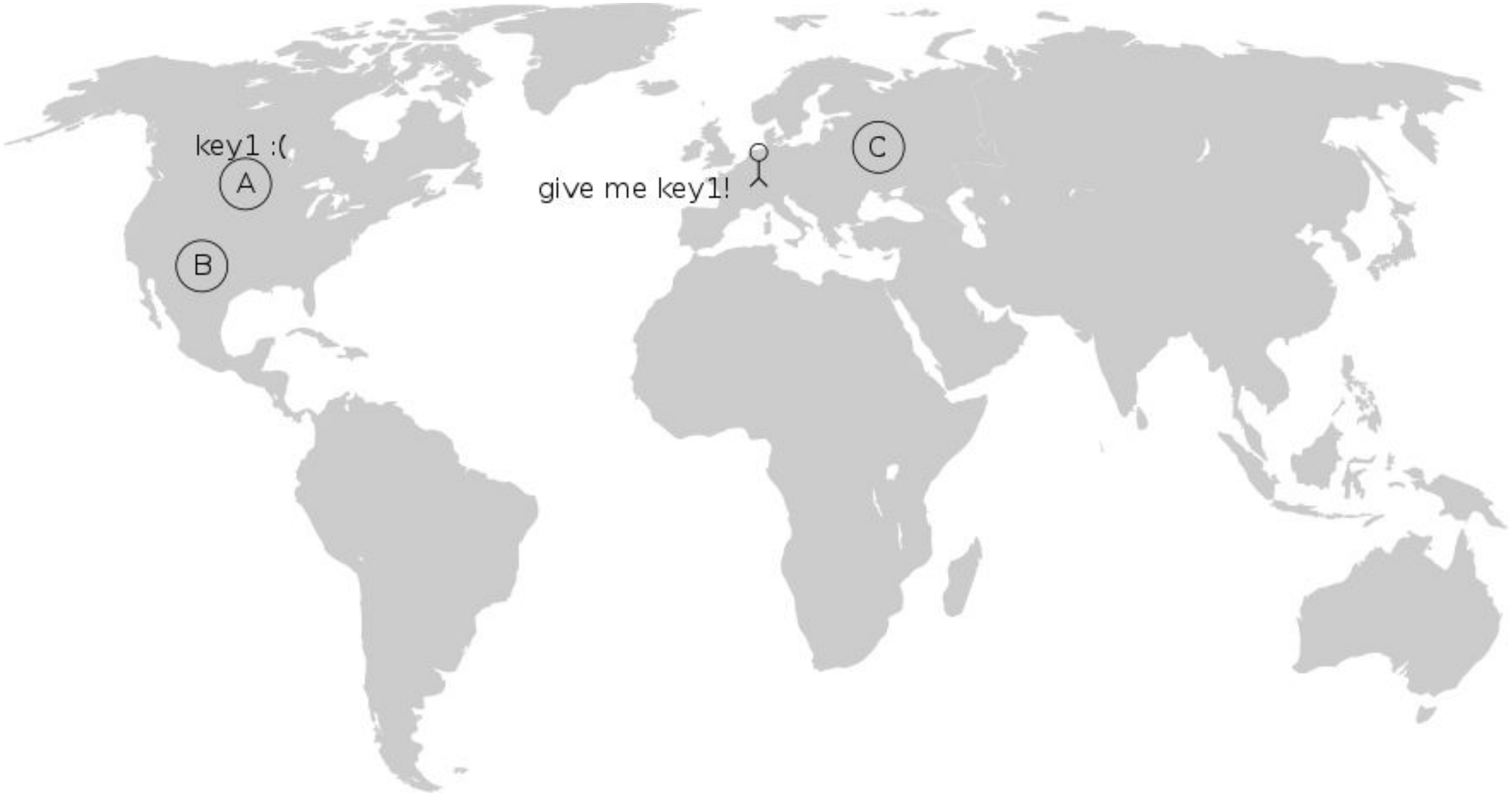
shards completely don’t care about data access locality :(

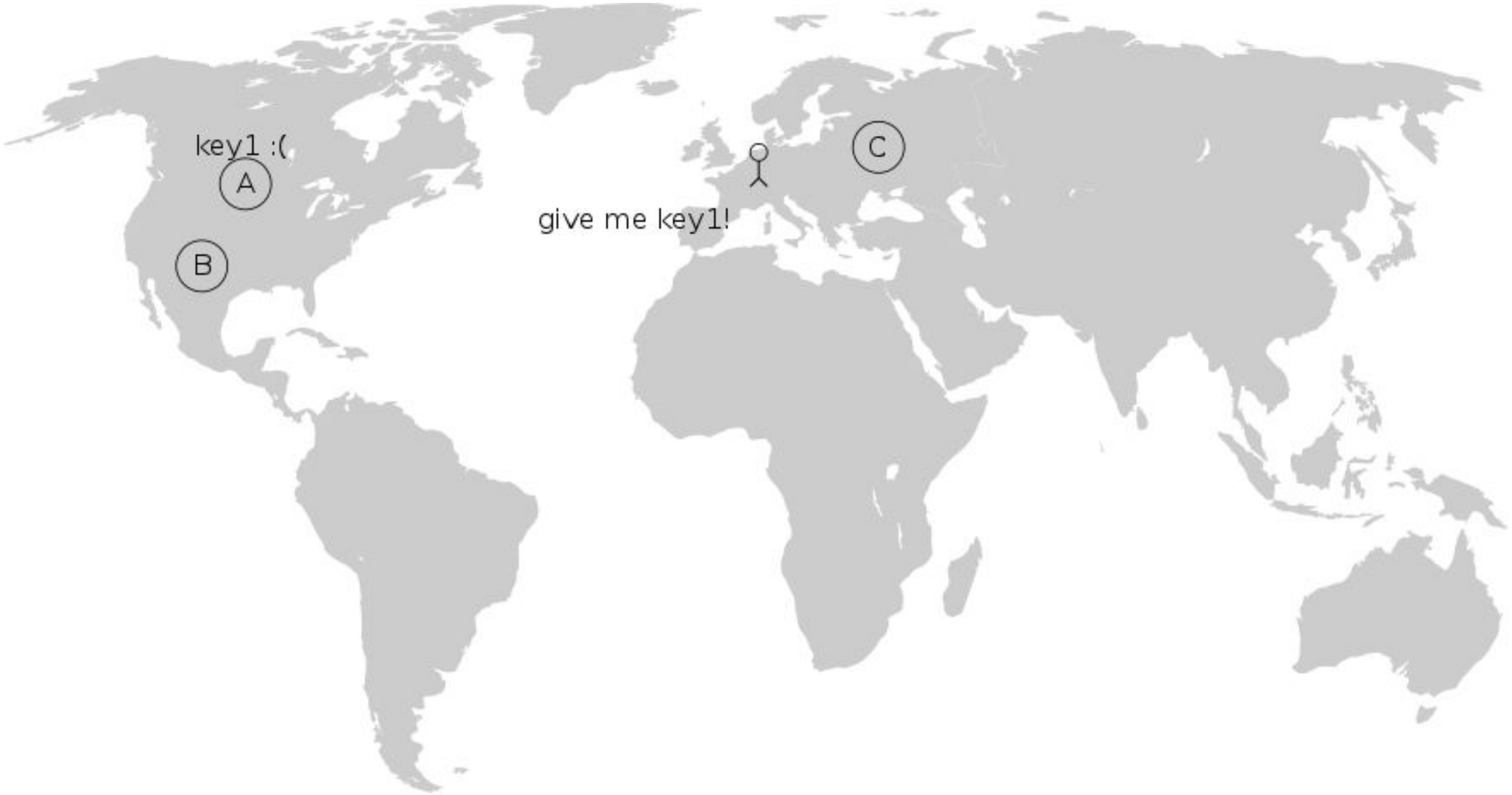
## Today’s solutions for data access locality

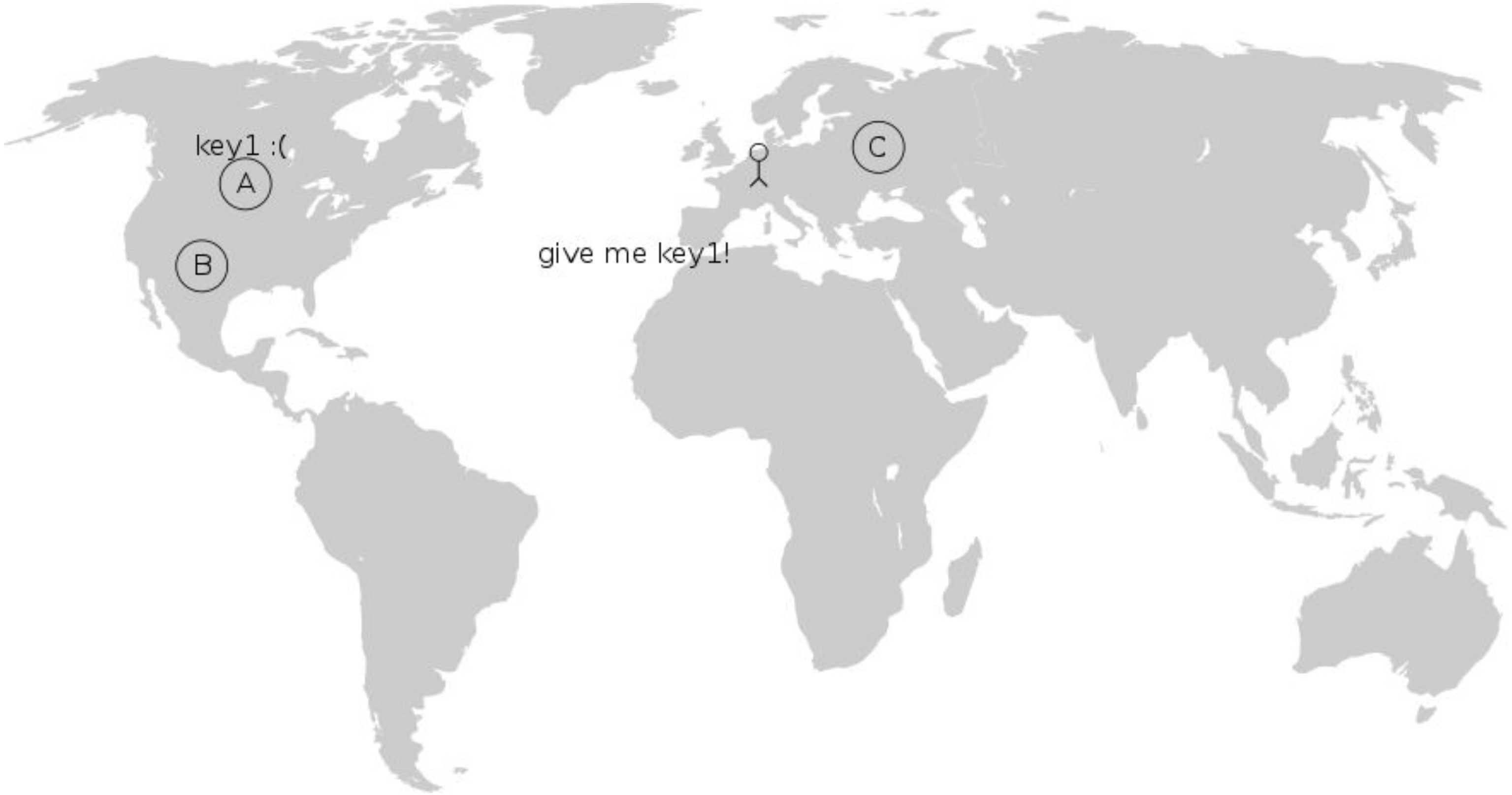
- distributed caches?
  - require high cache hit rates to be effective
  - for big DBs, this required significant hardware infrastructure
  - typically don’t offer strong consistency
- replicate everything to all DCs?
  - all reads are local!
  - but the storage overhead…
  - and writes become **very** costly

Neither is ideal, especially for datasets with low R/W ratios (e.g. 1).

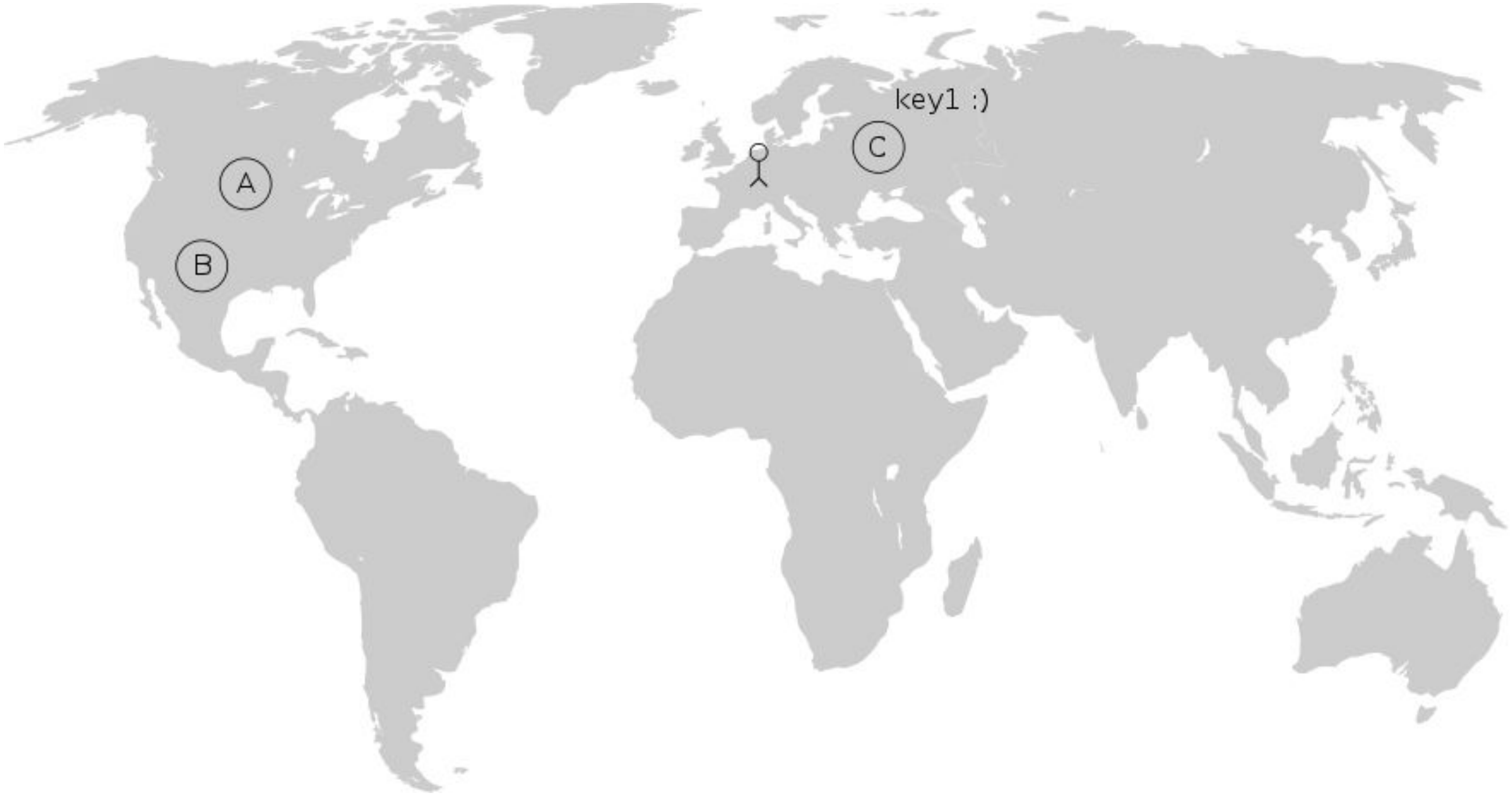
# Solution?




key1 :(
A
B
give me key1!
C


key1 :(
A
B
give me key1!
C

key1 :(
A
B
Akkio key1!
C


key1 :)
A
B
C

## Solution: data migration

- by the way, some systems have the option to migrate entire shards
- but due to their sizes, this is not effective for data access locality
- e.g. at Facebook, size of a typical working set of a single client: ~1MB

## Sharding the shards

- split our “big shards” into smaller datasets: **µ-shards**
- size: hundreds of bytes to few megabytes
- different purpose (solving data access locality)
- defined to serve this purpose
- easy to migrate dynamically when access patterns change
- different than shards:
  - “first-class citizen”: visible to the client application
  - the client, not the DB, defines how data is “µ-sharded”

Note: they don’t work for all datasets

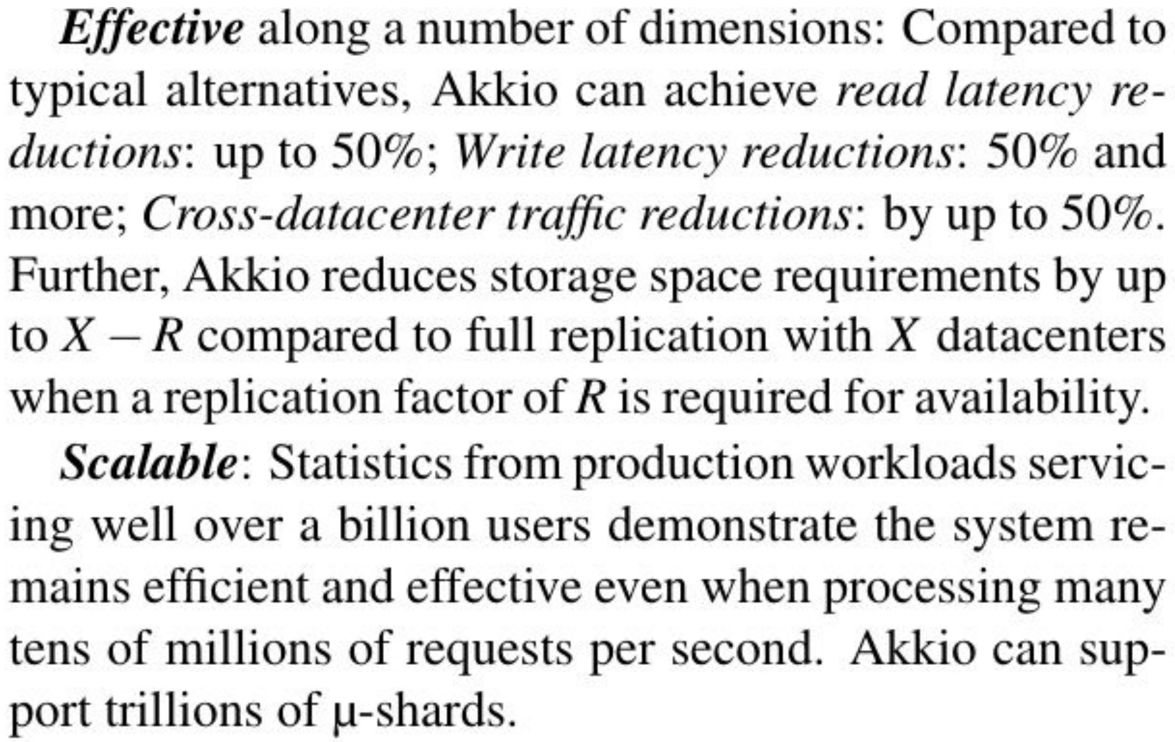
(e.g. Google Search, Facebook Social Graph)

## Akkio: a locality management service

A layer between the DB and the client application for managing μ-shards.

- The client defines how data is partitioned to μ-shards.
- Akkio:
    - maps μ-shards onto replica set collections,
    - tracks client accesses to μ-shards
    - migrates μ-shards when it thinks it’s a good idea
    - directs client access requests to appropriate replica sets

## 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.

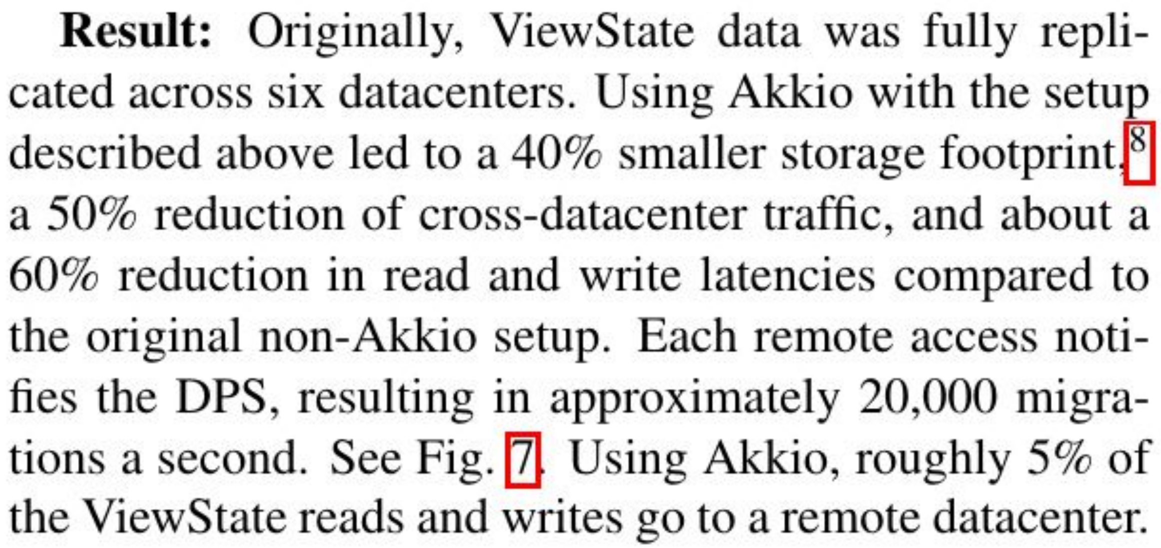
## ViewState

Example: the ViewState service (at FB)

- stores history of content previously shown to user
- each time user is shown content, new data is appended to ViewState
- used to decide what to show next

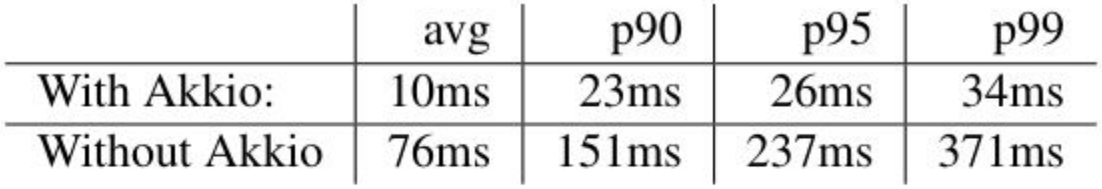
After installing Akkio at ViewState...

## ViewState



**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.

## Another example: AccessState



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

*Table 4: AccessState client service access latencies.*

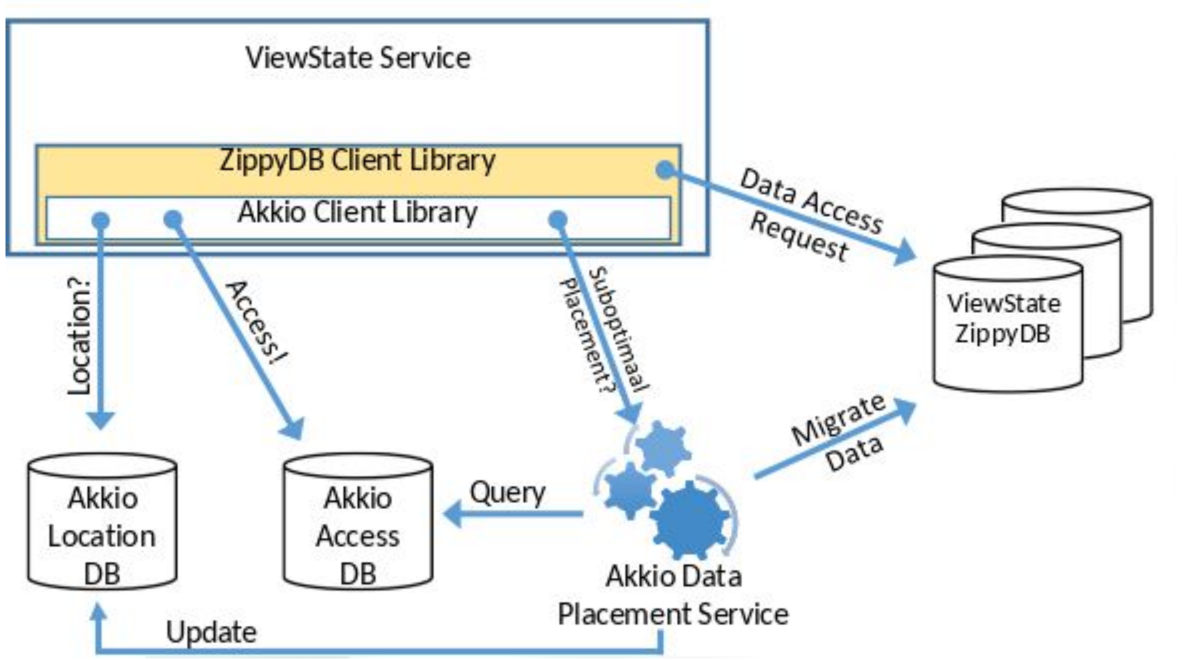
[30] PLUGGE, E., HOWS, D., MEMBREY, P., AND HAWKINS, T. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB*, 3rd ed. Apress, 2015.

[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.

## Akkio’s architecture



## Akkio’s architecture

- Akkio Location Service
  - knows where µ-shards are (maps them to replica set collections)
- Access Counter Service
  - counts accesses to all µ-shards
  - how many times, type (read/write), from where
- Data Placement Service
  - decides where to place µ-shards
  - manages migrations

(all services by themselves distributed, e.g. on top of ZippyDB)

## Accessing a key by the client

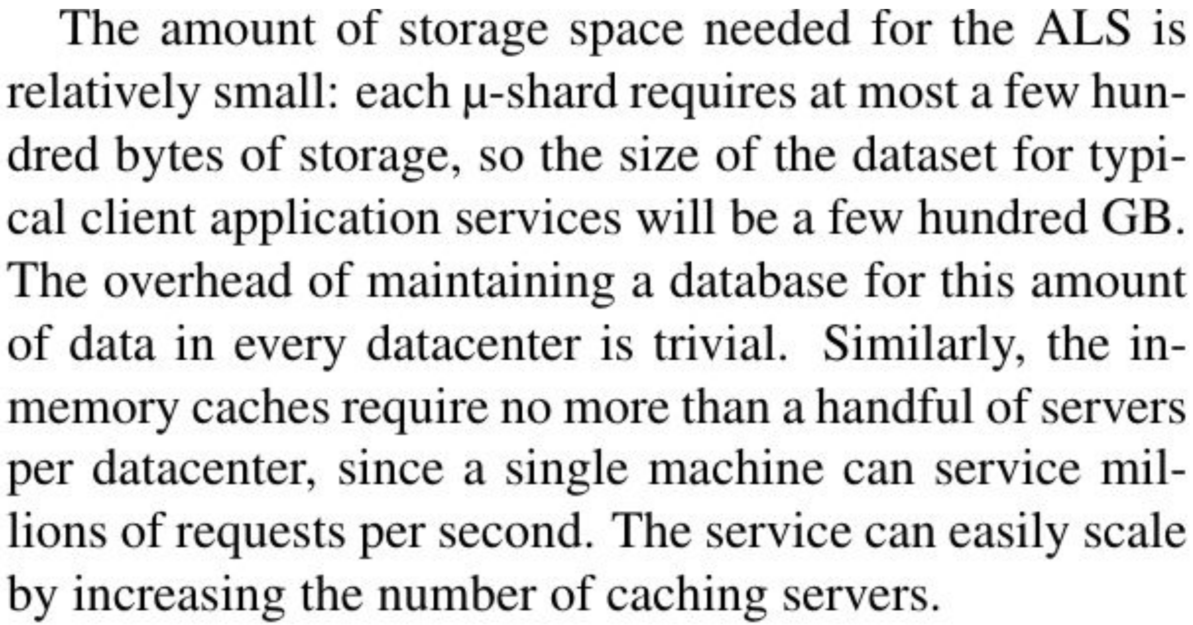
1. lookup the μ-shard of the key
2. ask ALS for the μ-shard's location

   (returns a location handle, identifying a replica set collection)
3. if the replication configuration of this collection has only remote DCs:
   a. notify DPS that perhaps a migration should be performed

      (this happens asynchronously: probably in a separate thread, perhaps sometime in the future)
4. contact a replica to access the data

ALS contacted on the critical path, so it uses distributed caches for speed.
Fortunately, storing locations of μ-shards is not very expensive...

## Sizes



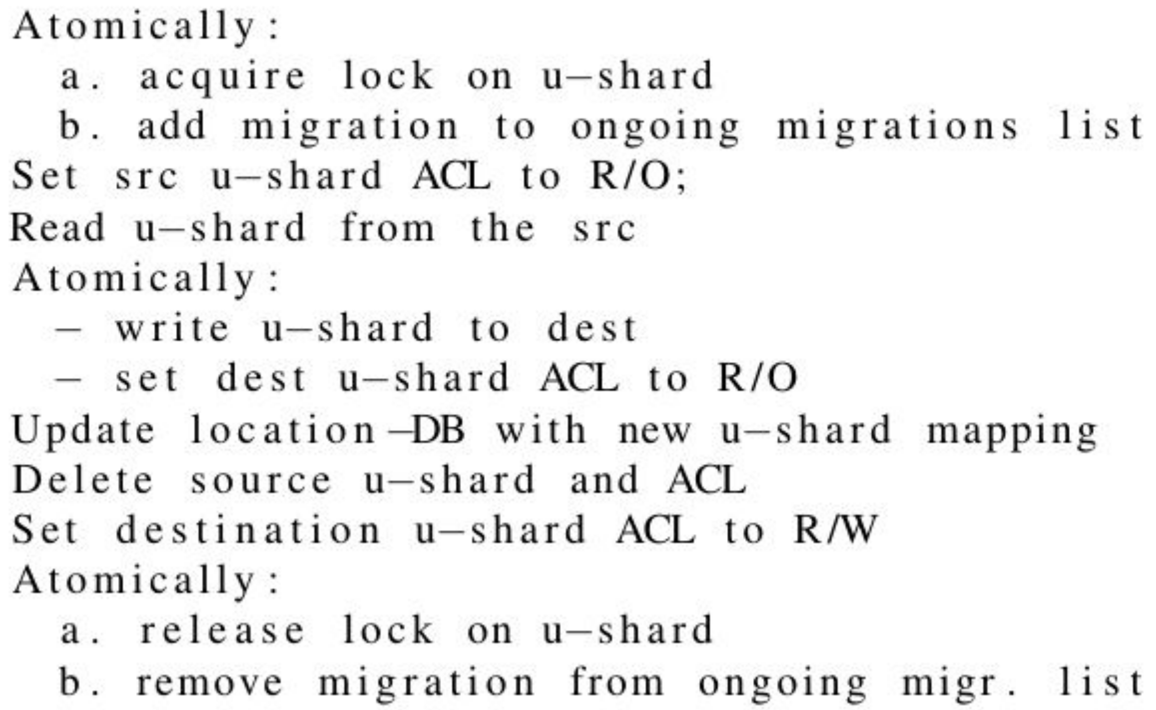
The amount of storage space needed for the ALS is relatively small: each μ-shard requires at most a few hundred bytes of storage, so the size of the dataset for typical client application services will be a few hundred GB. The overhead of maintaining a database for this amount of data in every datacenter 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.

## Migration

After client notices it had to do a remote data access to a μ-shard, it tells DPS.
DPS checks if it should migrate the μ-shard:

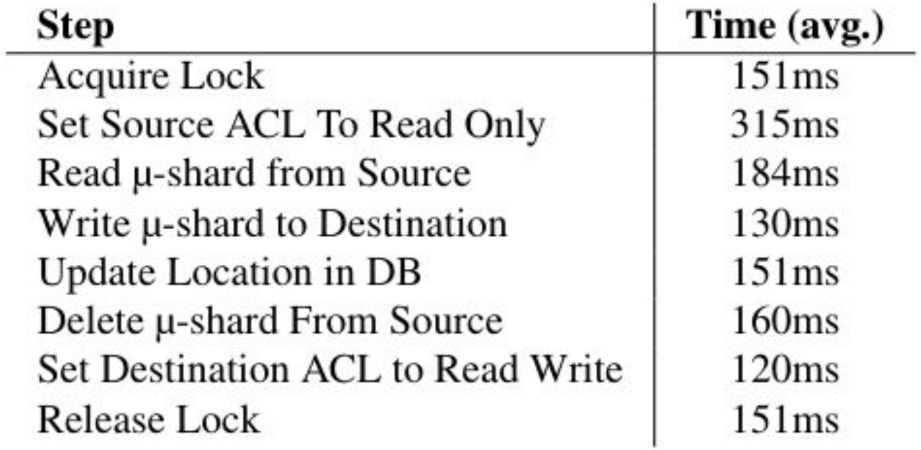
- asks ACS for the statistics
- calculates and assigns scores to replica set collections
- checks if the migration should be performed
- checks if the migration can be performed
- if so, migrates

## Migration



```
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
```

# Migration



| 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.*

## Fault recovery

What if a DPS instance crashes in the middle of a migration?

- every instance has a global sequence number
- the number is persisted with every state related to a pending migration
  e.g. with the μ-shard lock
- failed instances are restarted with a higher number
- restarted instance goes through a recovery process:
  - queries the location DB to identify ongoing migrations initiated by the failed instance
  - scans the μ-shard on source and destination to identify which steps have been completed

"the sequence number for recovered migration is updated to avoid any conflicts with a stale, failed DPS server instance"