# CS 5412/LECTURE 2
# ELASTICITY AND SCALABILITY

Ken Birman
Spring, 2019

# CONCEPT OF ELASTIC SCALABILITY

*Not this kind of elastic!*

The sheer size of the cloud requires a lot of resources. These are allocated *elastically*, meaning “on demand”.

Size: A company like Facebook needs to run data centers on every continent, and for the United States, has four major ones plus an extra 50 or so “point of presence” locations (mini-datacenters).

A single data center might deal with millions of simultaneous users and have many hundreds of thousands of servers.

# LOAD SWINGS ARE INEVITABLE IN THIS MODEL

People sleep in Seattle while they are waking up in New York.

People work in Mumbia when nobody is working in the USA

So any particular datacenter sees huge variation in loads.

# SHOULD THESE ELASTIC SERVICES BE BUILT AS <u>ONE BIG THING</u>, OR <u>MANY SMALLER THINGS</u>?

Much like our “how to implement a NUMA server topic”

**One big thing**: we would need to have all the members of the elastic pool share a single “state”, like by holding it in a database (or replicating data but holding it in memory).

**Many small things:** We split the data itself into chunks (“shards”).

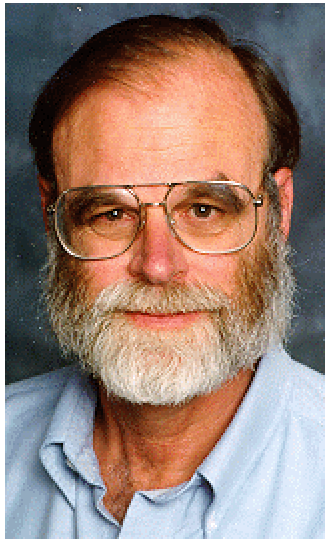
# SINGLE BIG SERVICE PERFORMS POORLY… WHY?

Until 2005 “one server” was able to scale and keep up, like for Amazon’s shopping cart.  A 2005 server often ran on a small cluster with, perhaps, 2-16 machines in the cluster.

This worked well.

But suddenly, as the cloud grew,this form of scaling broke.  Companies threw unlimited money at the issue but critical services like databases still became hopelessly overloaded and crashed or fell far behind.

# JIM GRAY'S FAMOUS PAPER



**Jim Gray**
**(Jan 1944 – Jan 2007)**

At Microsoft, Jim Gray anticipated this as early as 1996.

He and colleagues wrote a wonderful paper from their insights:

> **The dangers of replication and a solution.** Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. In Proceedings of the 1996 ACM SIGMOD Conference. pp 173-182. DOI=http://dx.doi.org/10.1145/233269.233330

Basic message: divide and conquer is really the *only* option.

# HOW THEIR PAPER APPROACHED IT

The paper uses a “chalkboard analysis” to think about scaling for a system that behaves like a database.

- It could be an actual database like SQL Server or Oracle
- But their “model” also covered any other storage layer where you want strong guarantees of data consistency.
- Mostly they talk about a single replicated storage instance, but look at structured versions too.

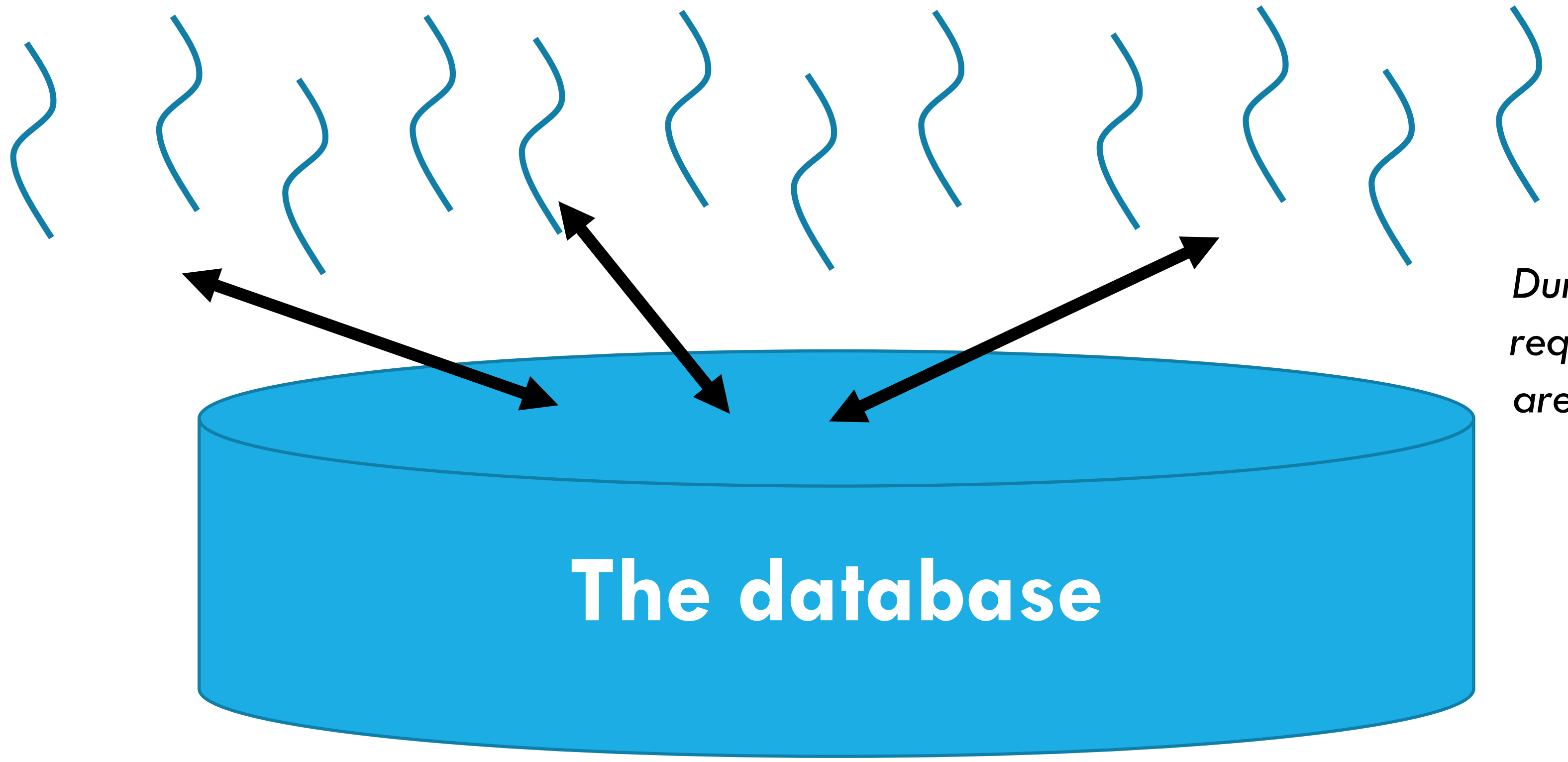
# THE CORE ISSUE

The paper assumes that your goal is some form of lock-based consistency, which they model as database serializability (in CS5412 we might prefer "state machine replication", but the idea is similar).

So applications will be using read locks, and write locks, and because we want to accommodate more and more load by adding more servers, the work spreads over the pool of servers.

This is a very common way to think about servers of all kinds.

# THEIR SETUP, AS A PICTURE

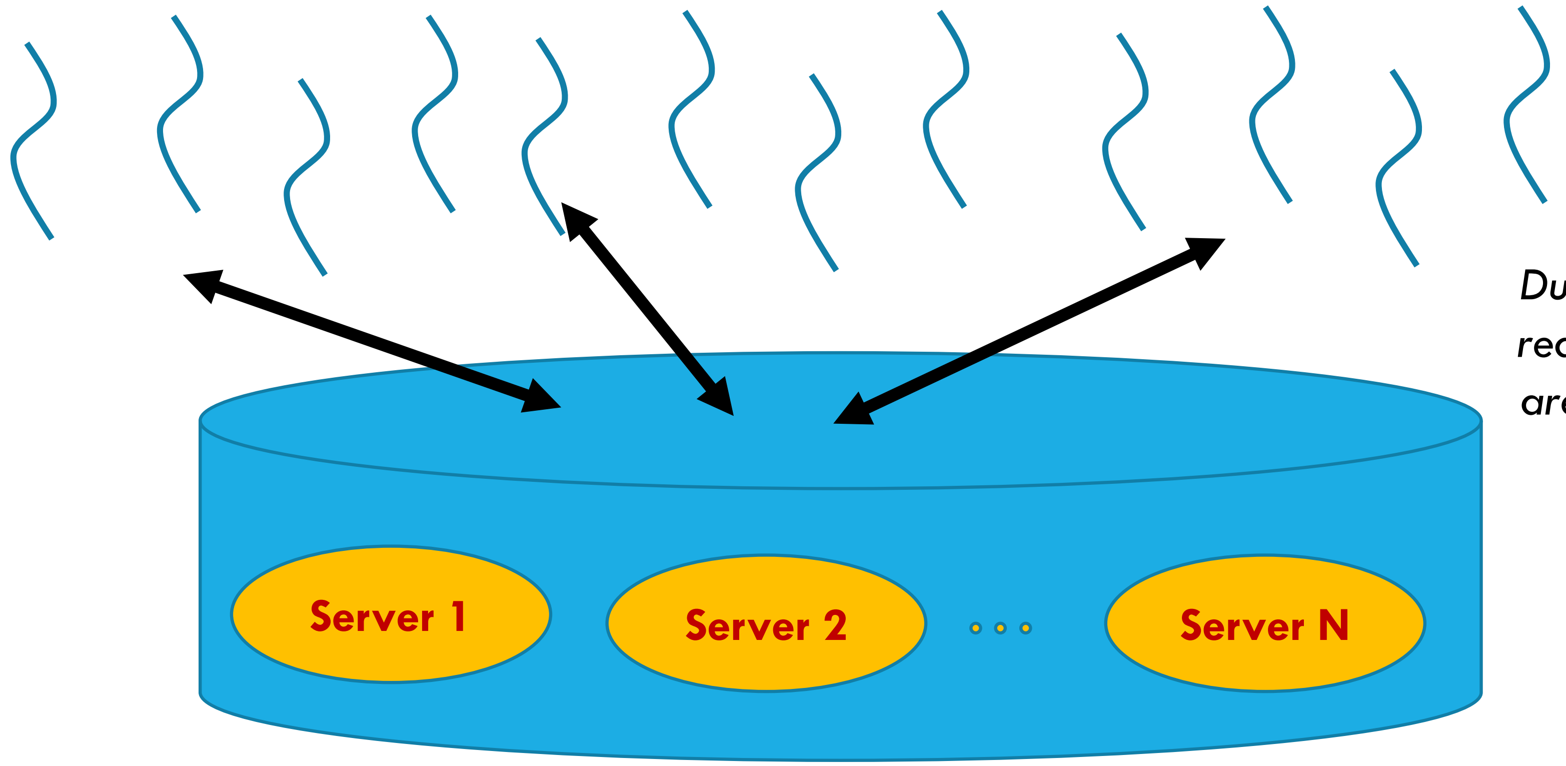



Applications using the database: *client processes*

*During the run, T concurrent requests are issued. Here, 3 are running right now, but T could be much larger.*

# THEIR BASIC SETUP, AS A PICTURE

Applications using the database: *client processes*




*During the run, T concurrent requests are issued. Here, 3 are running right now, but T could be much larger.*

*For scalability, the number of servers (N) can be increased*

# WHAT SHOULD BE THE GOALS?

A scalable system needs to be able to handle "more T's" by adding to N

Instead, they found that the work the servers must do will increase as $T^5$

Worse, with an even split of work, deadlocks occur as $N^3$, causing feedback (because the reissued transactions get done more than once).

Example: if 3 servers (N=3) could do 1000 TPS, with 5 servers the rate might drop to 300 TPS, purely because of deadlocks forcing abort/retry.

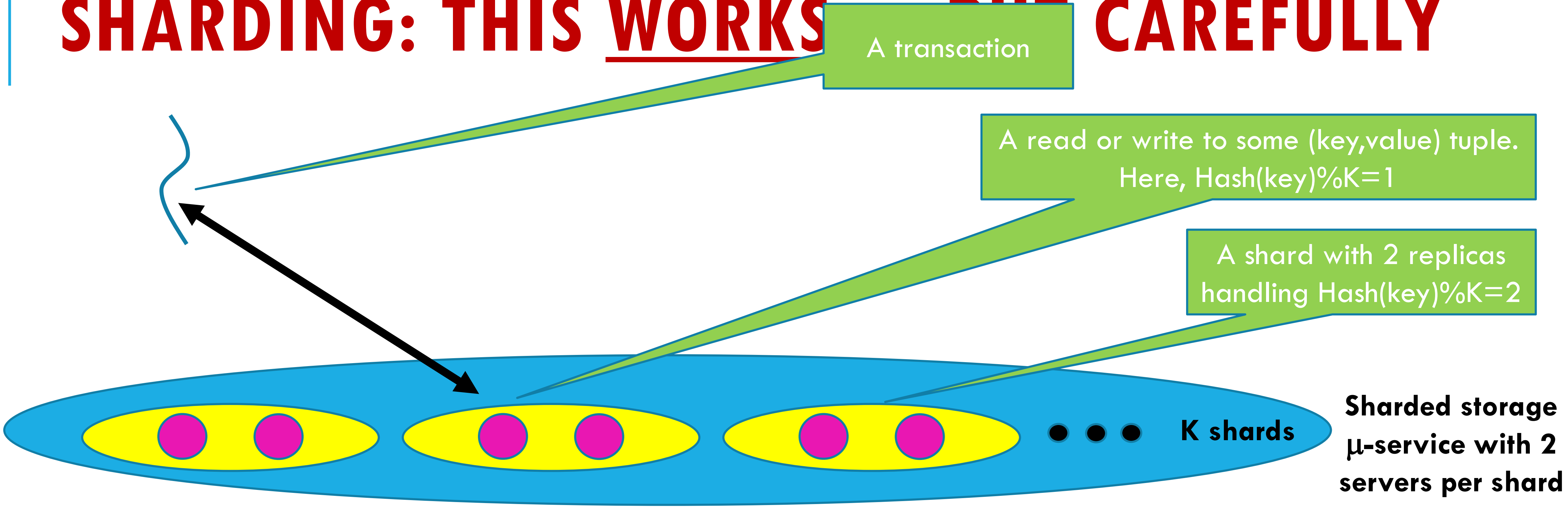
# WHY DO SERVICES SLOW DOWN AT SCALE?

The paper pointed to several main issues:

- Lock contention. The more concurrent tasks, the more likely that they will try to access the same object (birthday paradox!) and wait for locks.
- Abort. Many consistency mechanisms have some form of optimistic behavior built in. Now and then, they must back out and retry. Deadlock also causes abort/retry sequences.

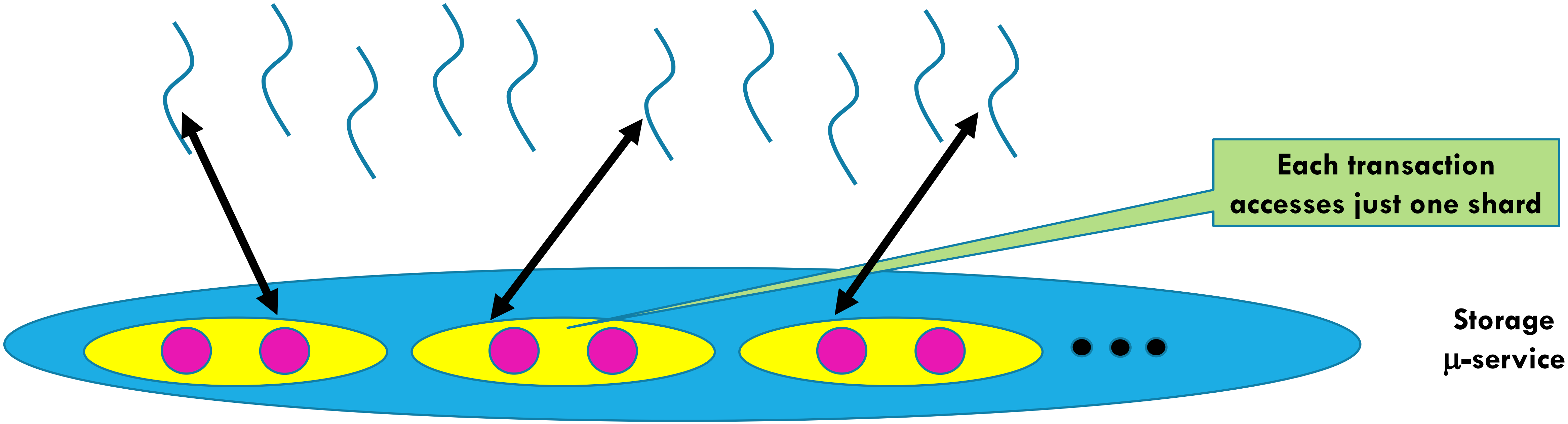
The paper actually explores multiple options for structuring the data, but ends up with similar negative conclusions *except in one specific situation*.

# SHARDING: THIS WORKS… BUT CAREFULLY




We will often see this kind of picture. Cloud IoT systems make very heavy use of key-based sharding. A (key,value) store holds data in the shards.

# SHARDING: THIS WORKS… BUT CAREFULLY




If a transaction does all its work at just one shard, never needing to access two or more, we can use *state machine replication* to do the work.

No locks or 2-phase commit are required.  This scales very well.

# DEFN: STATE MACHINE REPLICATION

We have a set of servers: {P, Q, R, … }

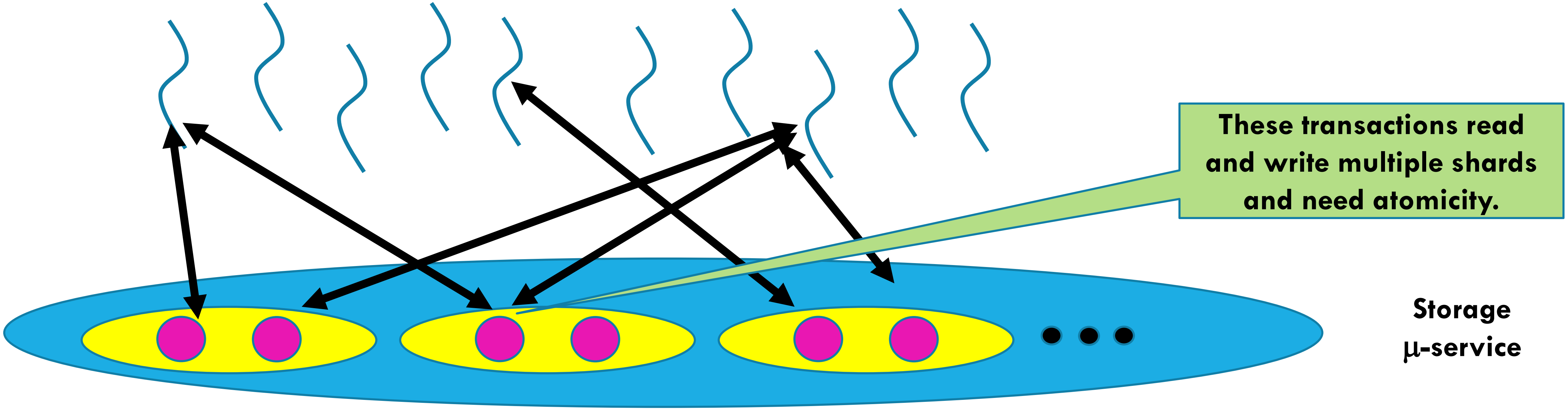
Each server holds a replica of some data and has the identical logic (code).

Any server can handle a read: the code just reads the local replica.

To do an update, we use an *atomic multicast* or a *durable Paxos write* (multicast if the state is kept in-memory, and durable if on disk).

Replicas see the same updates in the same order, hence stay in the same state.

# STATE MACHINE REPLICATION ON SHARDS DOESN'T SUPPORT ARBITRARY TRANSACTIONS




Transactions that touch multiple shards require complex locking. Jim Gray's analysis applies: as we scale this case up, performance collapses.

# STATE MACHINE REPLICATION IN WHICH EACH SHARD IS TOTALLY INDEPENDENT WORKS

If updates and queries are done entirely on one shard, Jim's analysis *does not apply*. This is because we can actually avoid locking (the update order is enough to ensure sequential progress of the replicas, with consistency).

An application could also access multiple shards, but independently, without any "cross shard" guarantees.

This is how sharded storage is used in today's cloud.

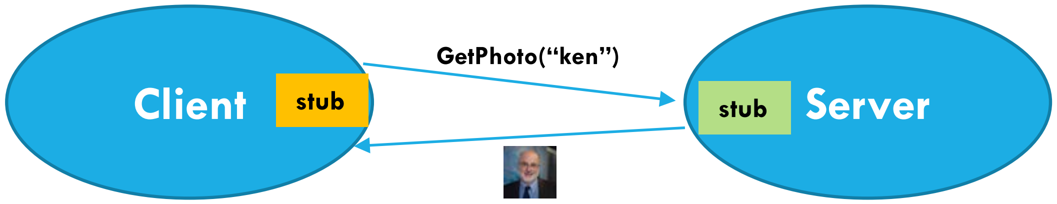
# EXAMPLE: A $\mu$-SERVICE FOR CACHING

Let's look closely at the concept of caching as it arises in a cloud, and at how we can make such a service elastic.

This is just one example, but in fact is a great example because key-value data structures are very common in the cloud.

Our example makes use of Google's GRPC: one of a few tools for helping client programs talk to server programs

# GOOGLE RPC (GRPC)




Server declares the callable procedures, "stubs" are created. These are methods with identical arguments. There is one each for client & server

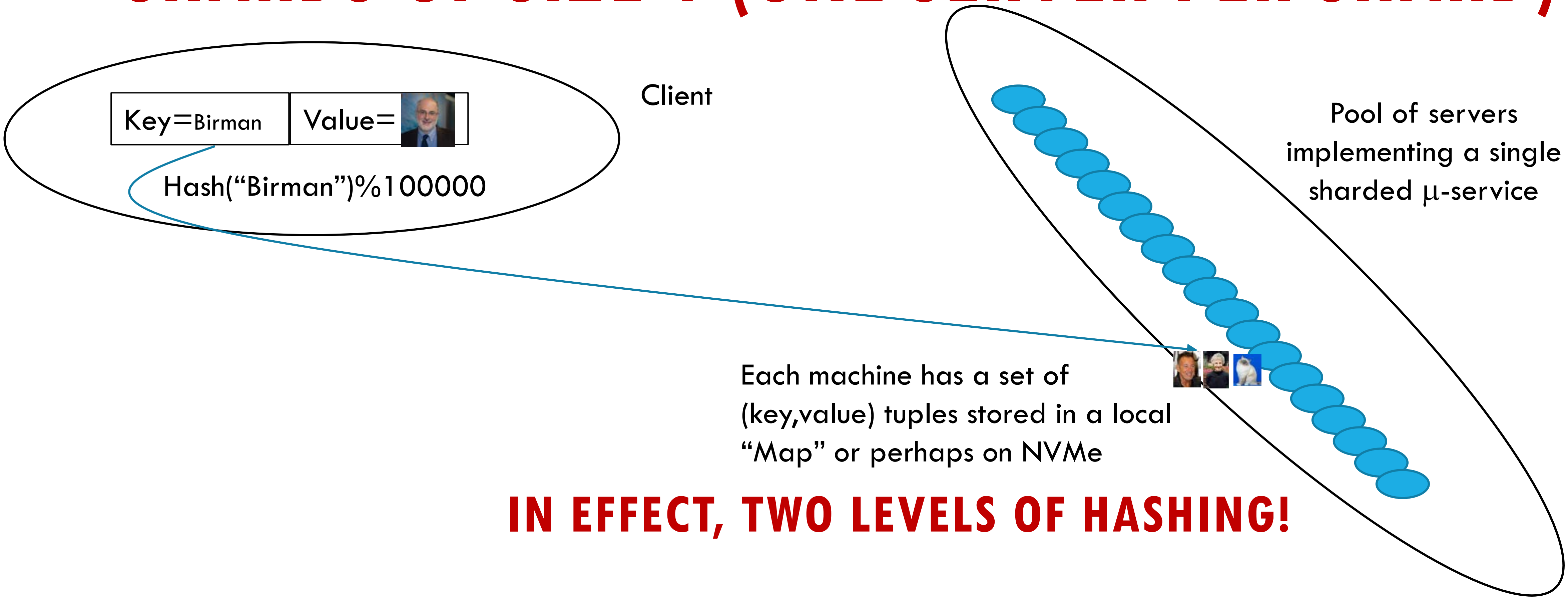
Client program links to the client stub. Client code can call **GetPhoto("ken")**

The stub starts by looking up server's network address ("binding"). If the server is running, the client makes a TCP connection.

Now the caller's arguments are *serialized* into a message, and it is sent on the TCP connection to the server. The server stub *deserializes* them.

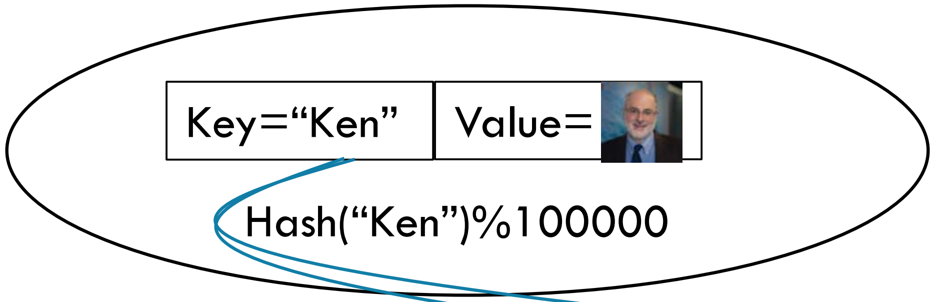
When the called procedure is done, a reply is sent back to the caller.

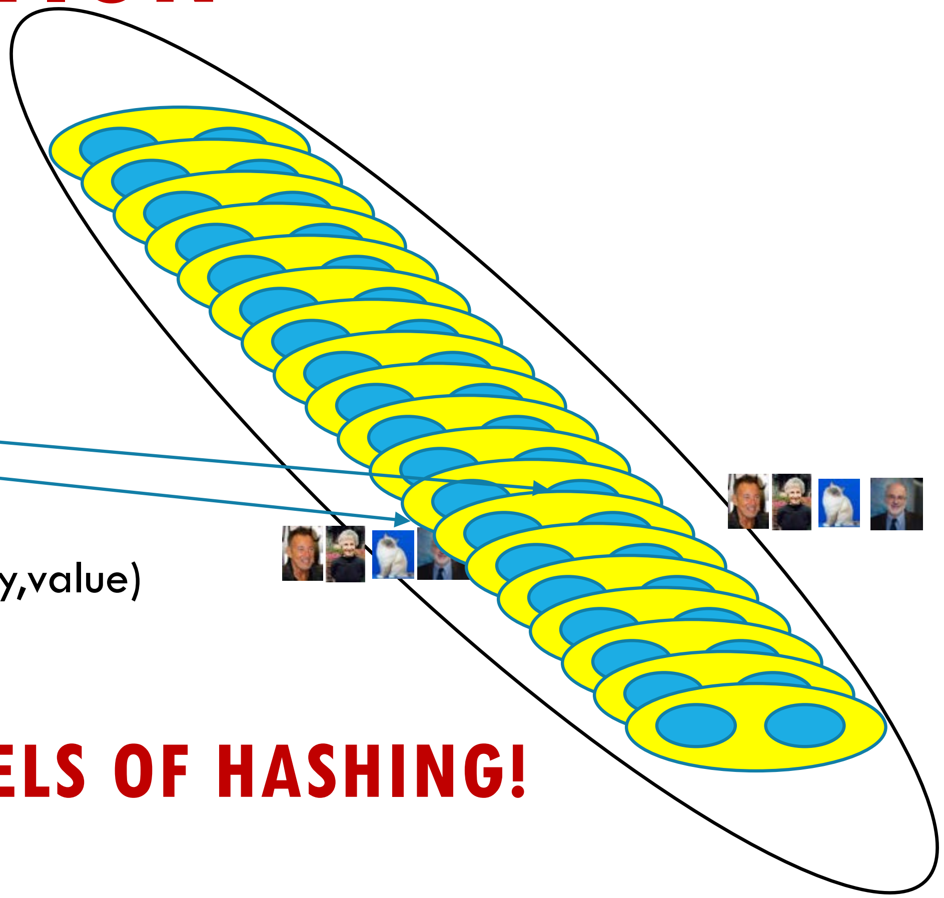
# ACCESSING SHARDED STORAGE, WITH SHARDS OF SIZE 1 (ONE SERVER PER SHARD)




## IN EFFECT, TWO LEVELS OF HASHING!

# WITH TWO SERVERS PER SHARD, WE ADD STATE MACHINE REPLICATION



These two machines both store a copy of the (key,value) tuple in a local "Map" or perhaps on NVMe

## IN EFFECT, TWO LEVELS OF HASHING!

# TERMINOLOGY

Our service in this example is called a “key value store” (KVS) or a “distributed hash table” (DHT)

Each replica holds a “shard” of the KVS: a distinct portion of the data.

Hashing is actually done using a cryptographic function like SHA 256.

# ELASTICITY ADDS A FURTHER DIMENSION

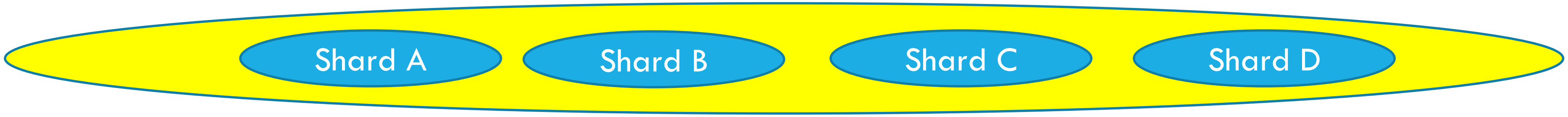
If we expect changing patterns of load, the cache may need a way to dynamically change the pattern of sharding.

Since a cache "works" even if empty, we could simply shut it down and restart with some other number of servers and some other sharding policy. But cold caches perform very badly.

Instead, we would ideally "shuffle" data.

# ELASTIC SHUFFLE

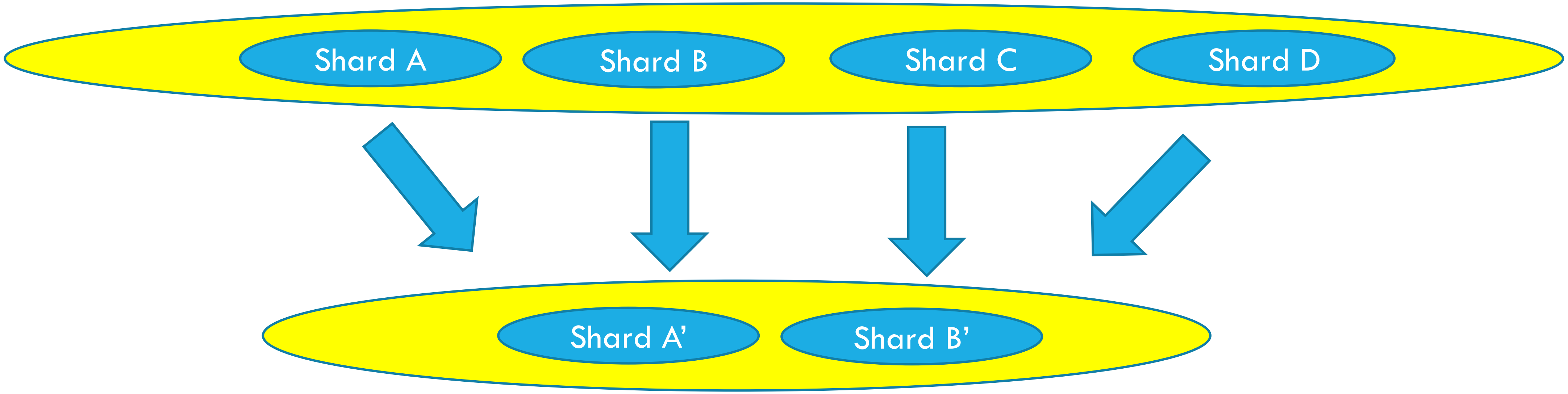
Perhaps we initially had data spread over 4 shards.




We could drop down to 2 shards during low-load periods. Of course half our items (hopefully, less popular ones) are dropped.

# ELASTIC SHUFFLE

Here, we shuffle data to map from 4 shards down to 2.




… later, we could shuffle it again to elastically expand the cache.

# BUT HOW WOULD OTHER SERVICES KNOW WHICH SERVERS OWN WHICH SHARDS?

A second issue now arises: how can applications that use the cache find out that the pattern just changed?

Typically, big data centers have a *management infrastructure* that owns this kind of information and keeps it in files (lists of the processes currently in the cache, and the parameters needed to compute the shard mapping).

If the layout changes, applications are told to reread the configuration. Later we will learn about one tool for this (Zookeeper). GRPC would consult this configuration data to find the server that the client needs to talk to.

# TYPICAL DHT API?

The so-called MemCached API was the first widely popular example. We saw it on Christina's slides on Tuesday.

Today there are many important DHTs (Cassandra, Dynamo DB,MemCached, and the list just goes on and on).

All support some form of (key,value) **put**, **get**, and (most) offer **watch.**

Some hide these basic operations behind file system APIs, or "computer-to-computer email" APIs (publish-subscribe or queuing), or database APIs.

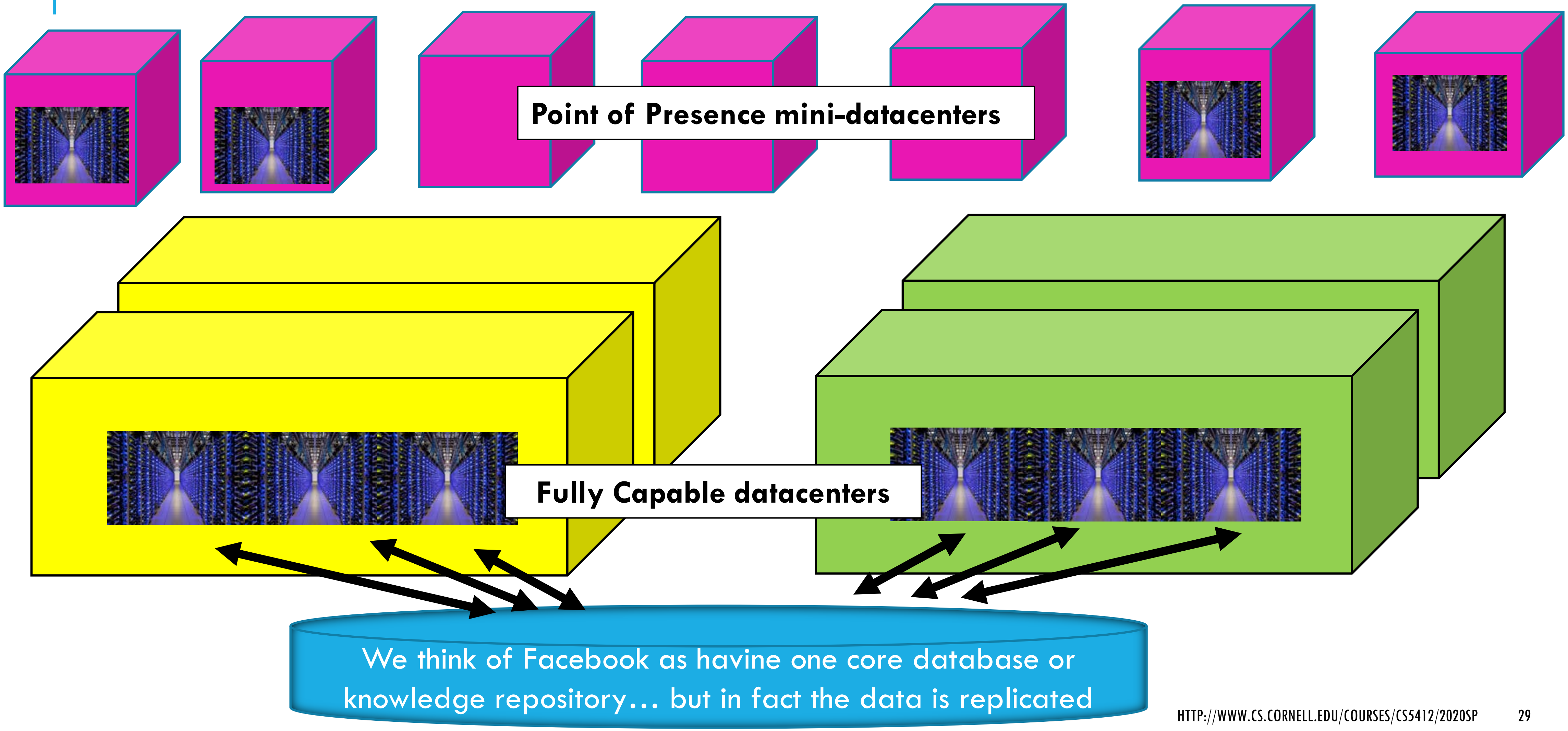
# USE CASE: FB CONTENT DELIVERY NETWORK

Facebook's CDN is a cloud-scale infrastructure that runs on point of presence datacenters in which key-value caches are deployed.

Role is to serve videos and images for end-users. Weak consistency is fine because videos and images are *immutable* (each object is written once, then read many times).

Requirements include speed, scaling, fault-tolerance, self-management

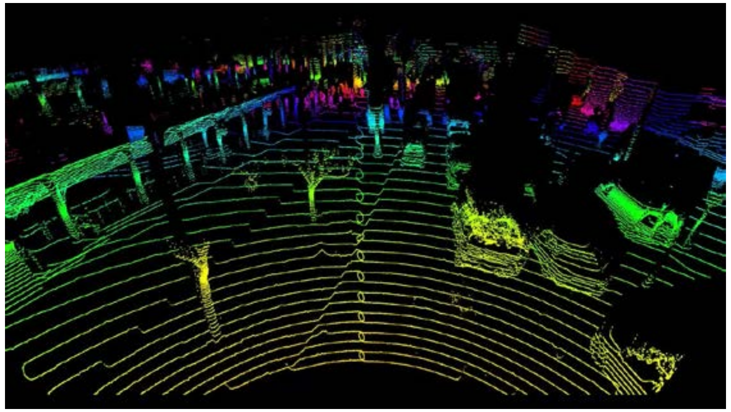
# THE FB BLOB CACHE IS PART OF A HIERARCHY DEPLOYED AT GLOBAL SCALE…



# … THE DATA IS JUST "BLOBS"

Facebook image data is stored in "blobs": Binary Large Objects

- This includes original images, videos
- Resized versions, and ones with different playback quality
- Versions that have been processed to tag people, *augmented reality*

# HAYSTACK

Holds the real image and video data in huge “film strips”, write-once.

Designed to retrieve any object with a single seek and a single read. Optimized for SSD (these have good transfer rates but are best for write-once, reread many loads, and have a long delay for starting a write).

Facebook doesn’t run a lot of copies

- One on the West Coast, one more on the East Coast
- Each has a backup right next to it.

Main issue: Haystack would easily get overloaded without caching

# A CACHE FOR BLOBS?


Goal: a μ-service we can run anywhere, and that can adapt in size as loads vary.

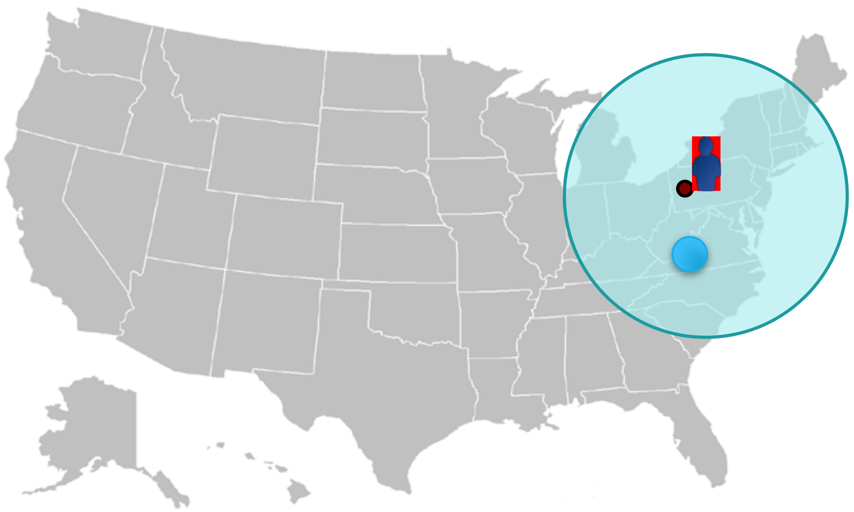
The keys would be photo or video id's.

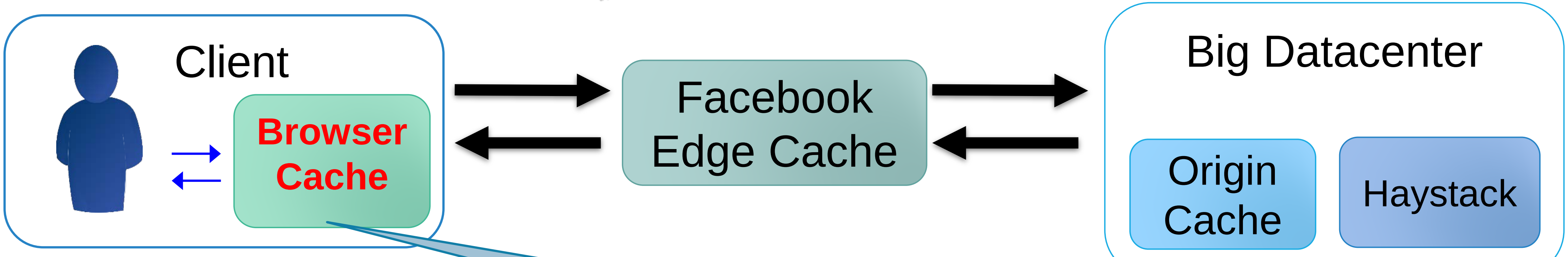
For each unique blob, FB has a special kind of tag telling us the resized dimensions of the particular instance we are looking at.

Resizing takes time and requires a GPU. So FB wants to avoid recomputing them.

# True data center holds the original photo in Haystack

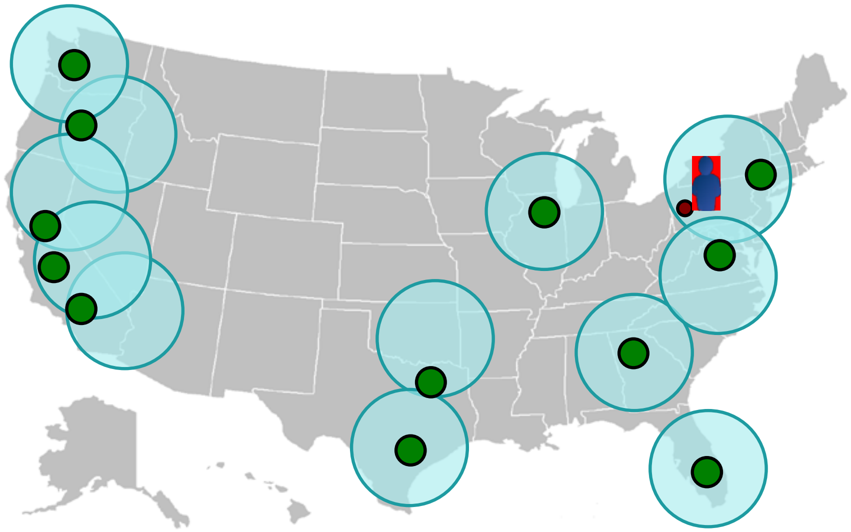
First, FB fetches your feed. This will have URLs with the image ID embedded in them. The browser tells the system what size of screen it has.

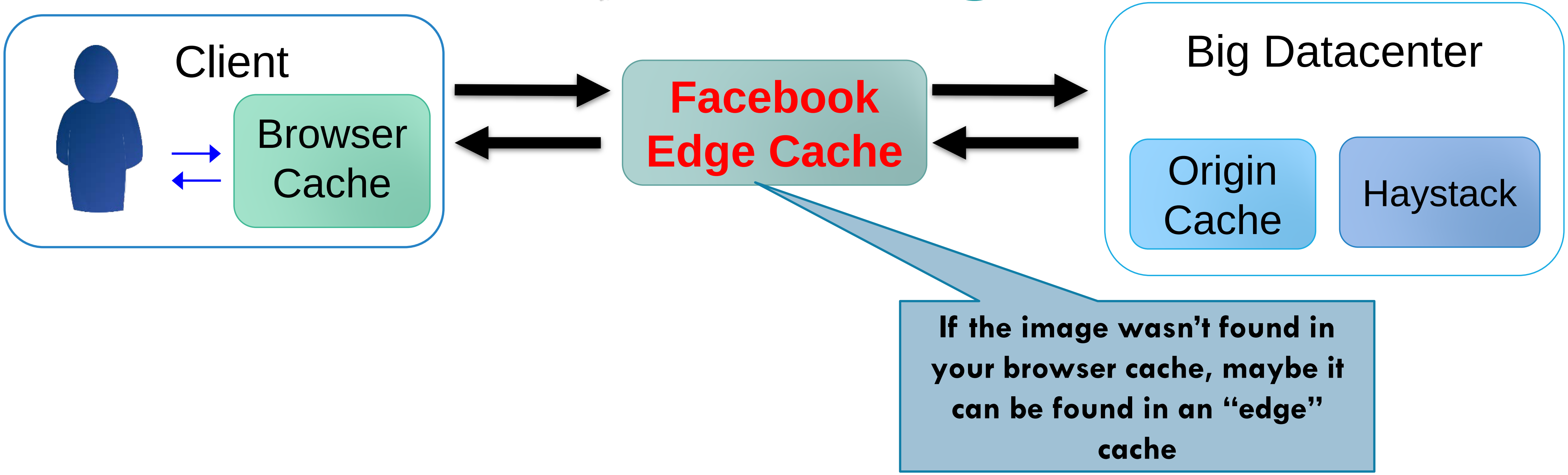





If you've recently seen the image, Facebook finds the blob in a cache on your computer

With a UID and a target size, we can search for the blob in the nearest point of presence cache.





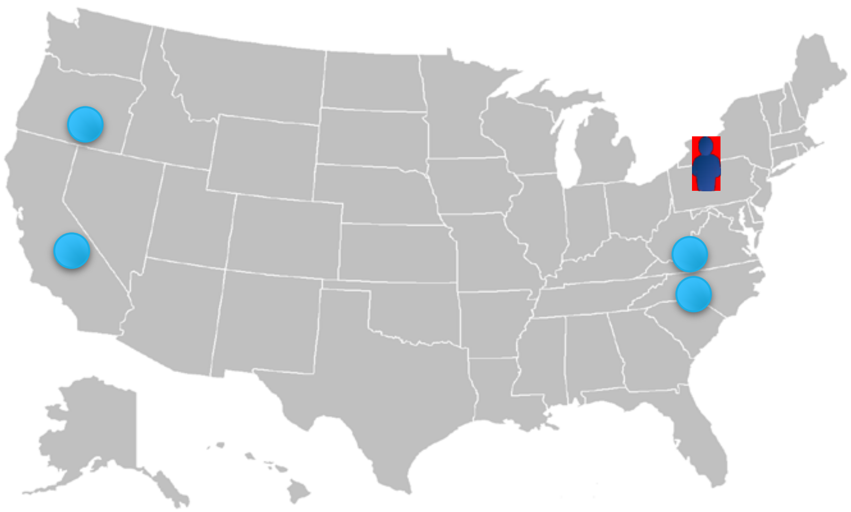
Origin:

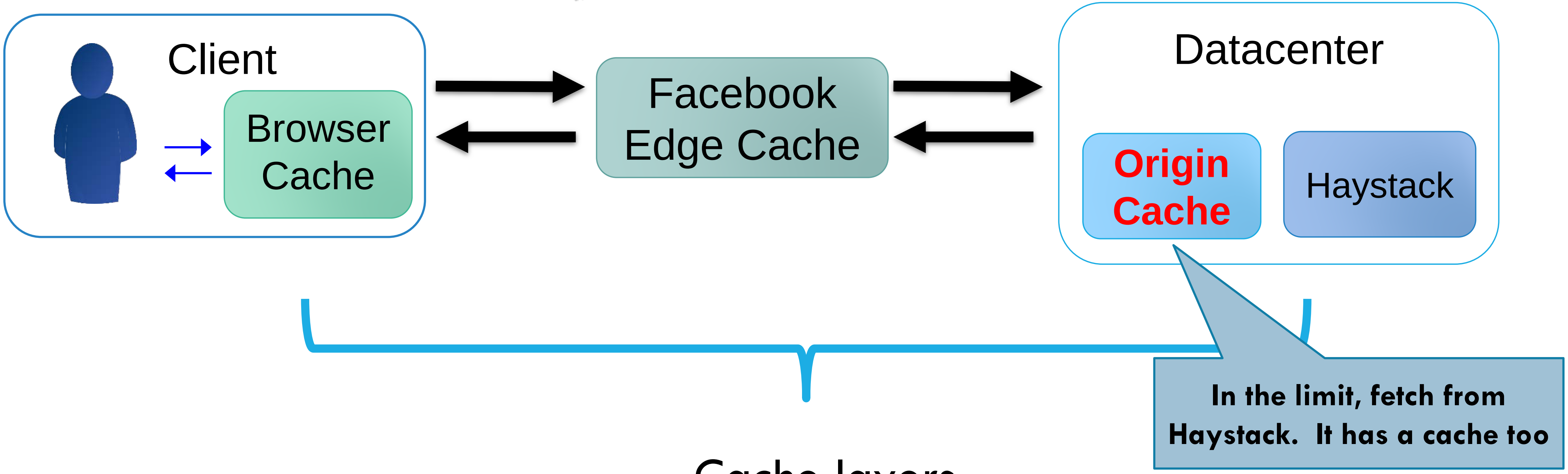
Coordinated

FIFO

Main goal:

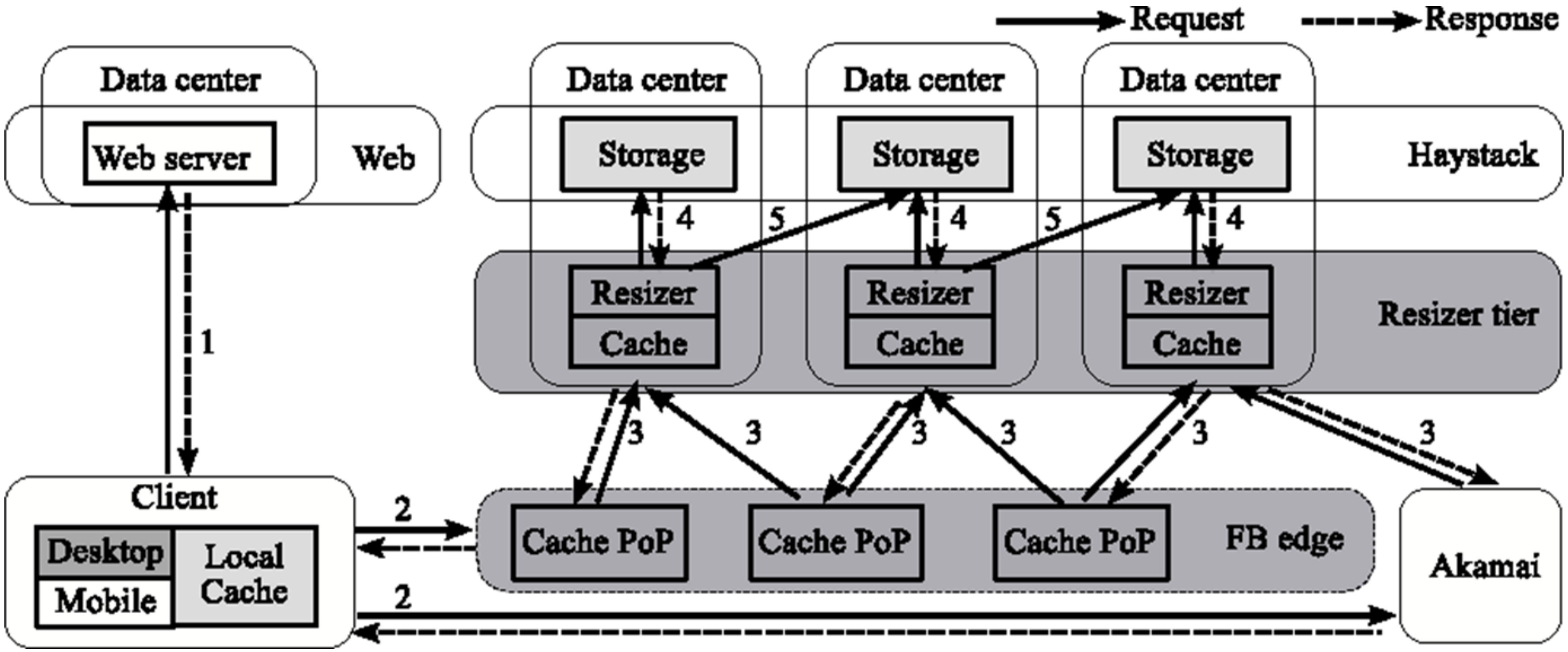
traffic sheltering



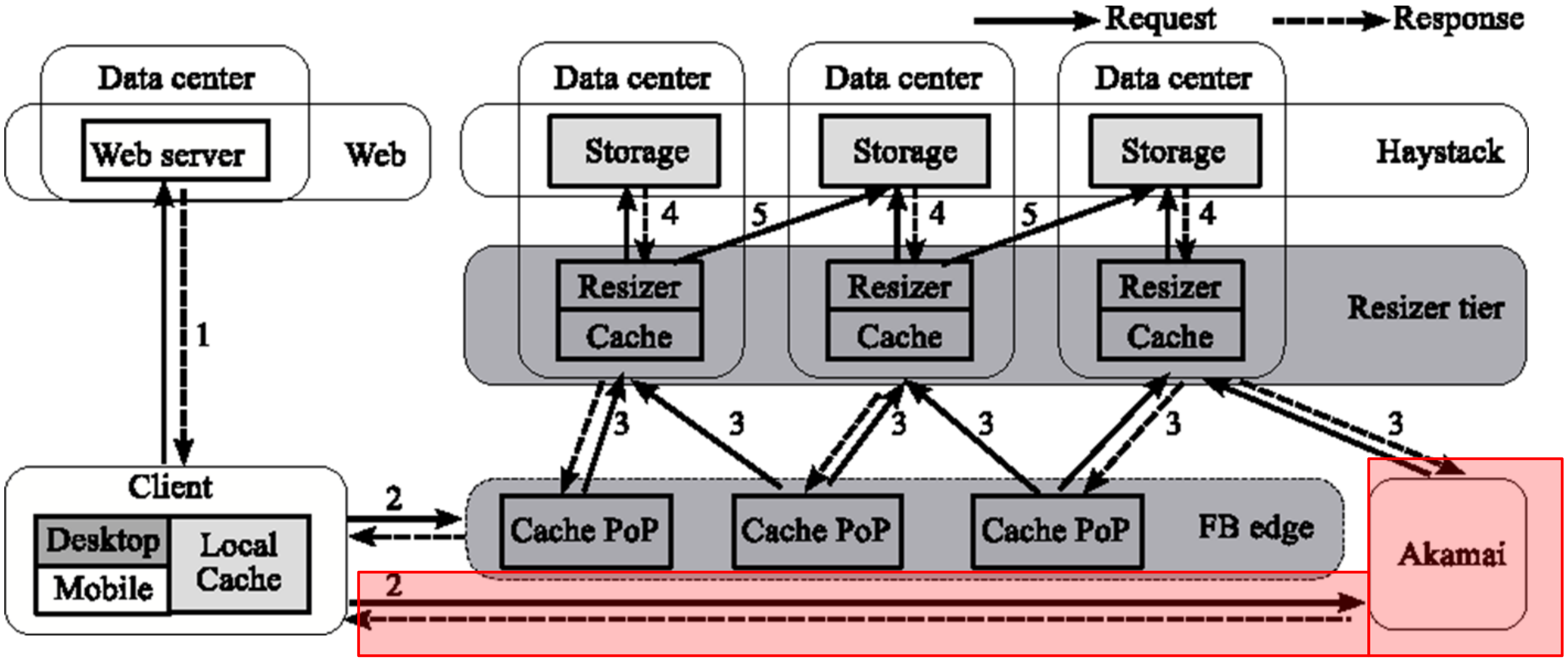



Cache layers

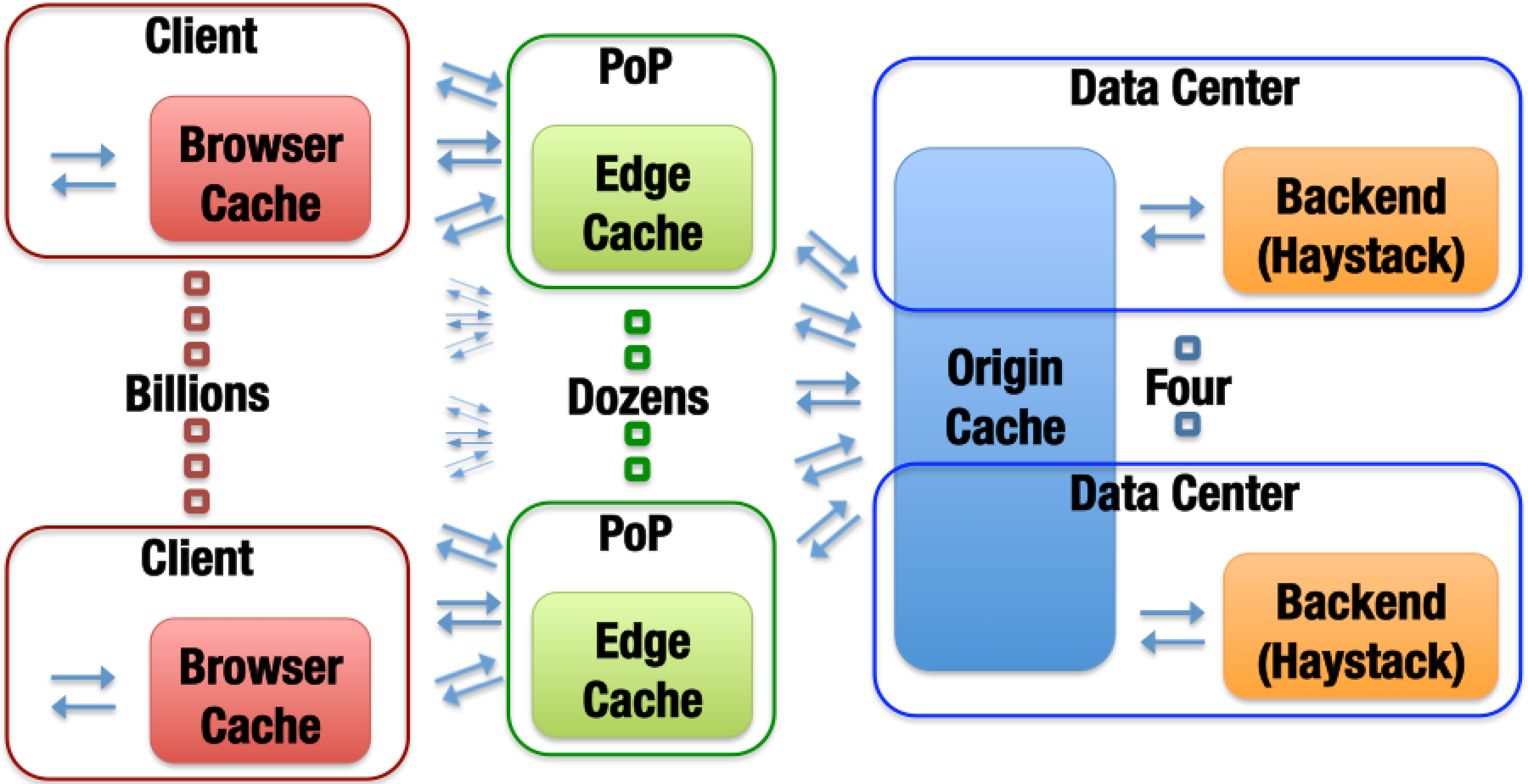
# ARCHITECTURAL DETAIL (COMPLEX)
# DARK GRAY: WE INSTRUMENTED IT
# PALE GRAY: WE CAN FIGURE OUT ITS BEHAVIOR



# ARCHITECTURAL DETAIL (WE WEREN'T ABLE TO INSTRUMENT THE PINK PART)

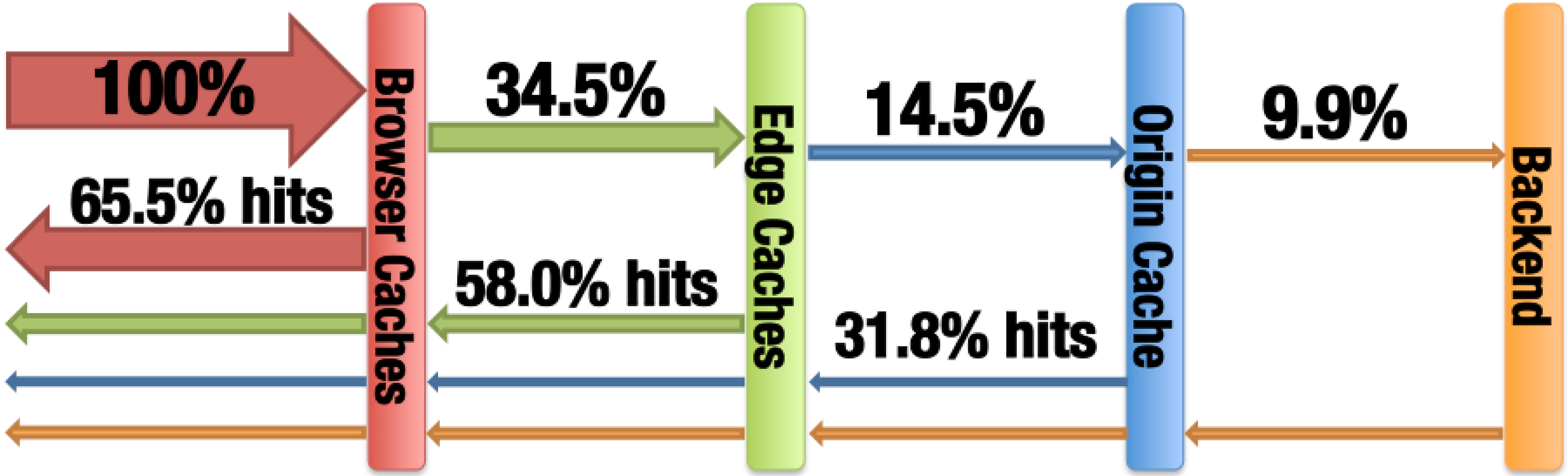


# BLOB-SERVING STACK (THE FB PORTION)



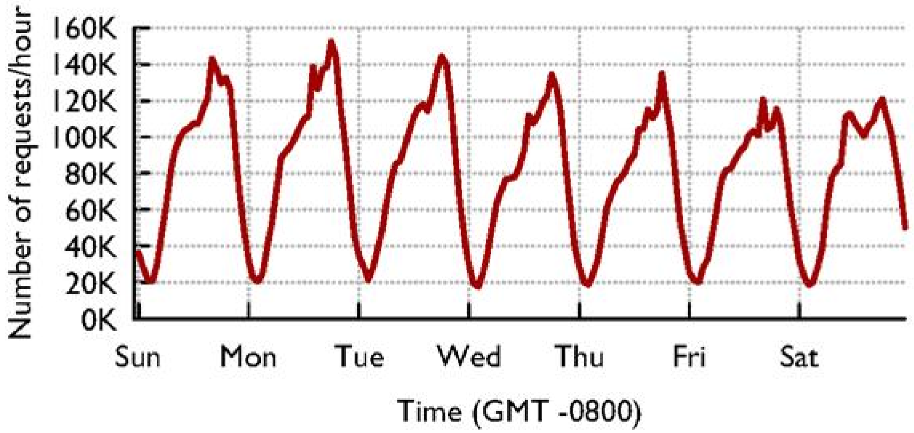
# WHAT WE OBSERVED

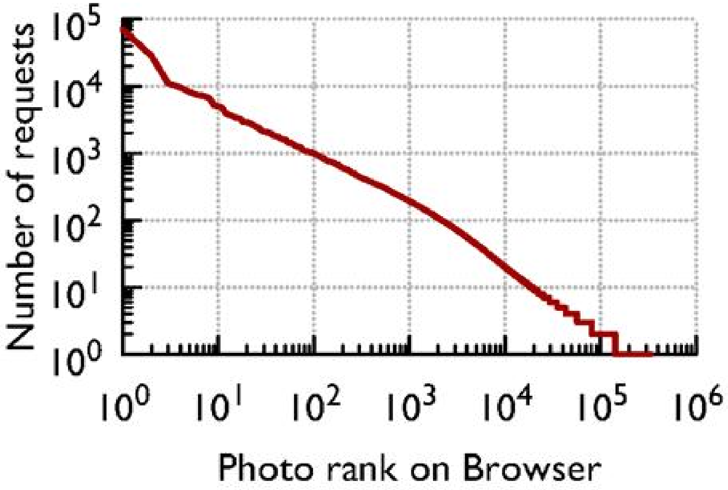
Month long trace of photo accesses, which we sampled and anonymized. Captures cache hits and misses at every level.



# CACHES SEE "CIRCADIAN" PATTERNS

Accesses vary by time of day…



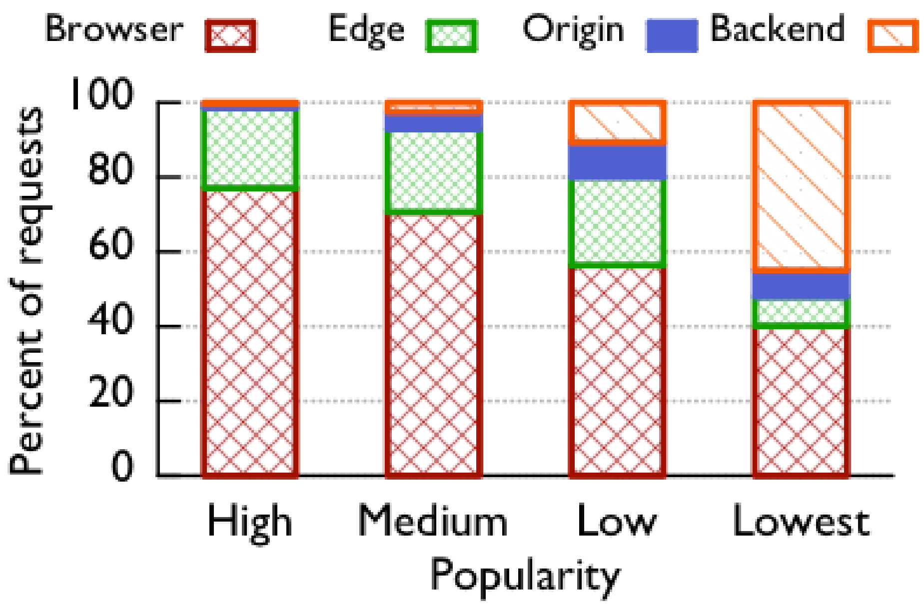




… and by photo: Some are far more popular than others

# PHOTO CACHEABILITY BY POPULARITY

The main job of each layer is different.

This is further evidence that cache policy should vary to match details of the actual workload



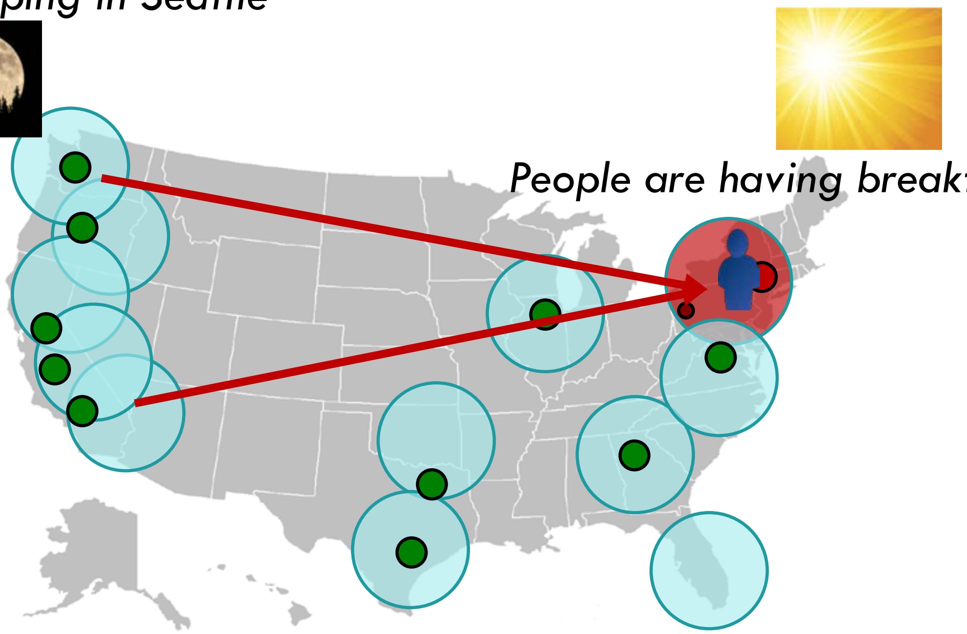
# GEO-SCALE CACHING

One way to do far better turns out to be for caches to collaborate at a WAN layer – some edge servers may encounter "suddenly popular" content earlier than others, and so those would do resizing operations first.

WAN collaboration between Edge caches is faster than asking for it from Haystack, and also reduces load on the Haystack platform.

Key insight: the Facebook Internet is remarkably fast and stable, and this gives better than scaling because the full cache can be exploited.

Remote fetch: cooperation between point-of-presence cache systems to share load and resources


In Ithaca, most of your content probably comes from a point of presence in the area, maybe the red one (near Boston)

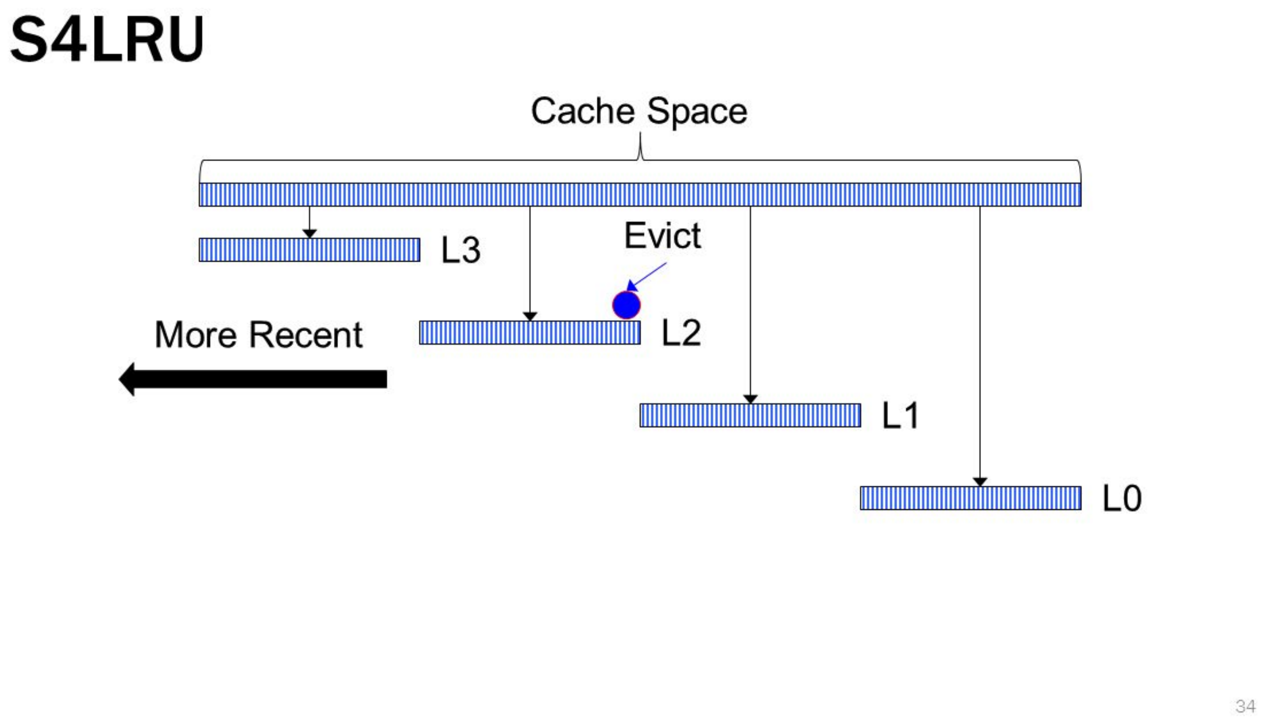
But if Boston doesn't have it or is overloaded, they casually reach out to places like Spokane, or Arizona, especially during periods when those are lightly loaded, like 5am PT!

# CACHE RETENTION/EVICTION POLICY

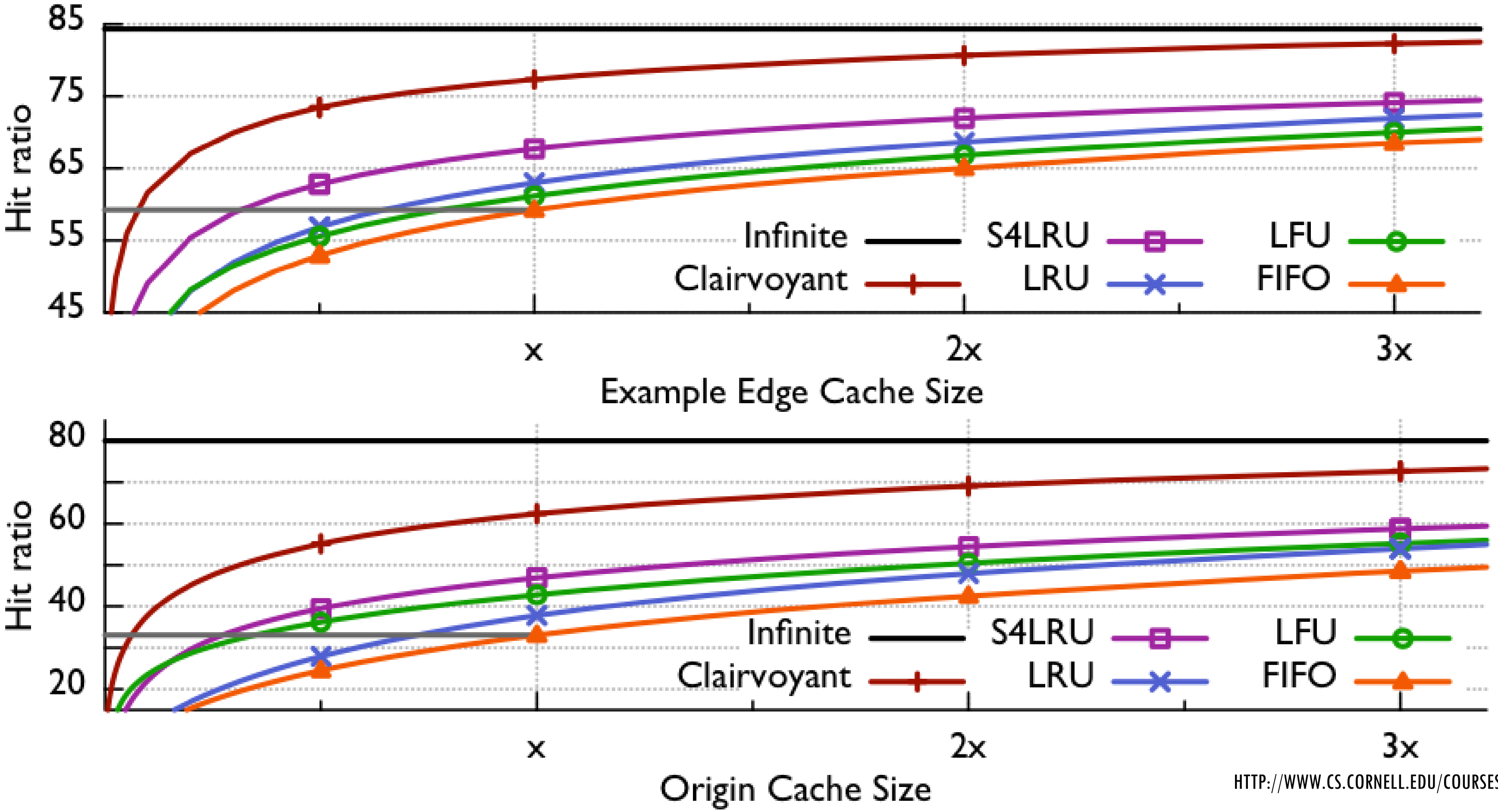
Facebook was using an LRU policy.

We used our trace to evaluate a segmented scheme called S4LRU

It outperforms all other algorithms we looked at



# HERE WE SEE THAT S4LRU IS FAR BETTER THAN NORMAL LRU



# SO… SWITCH TO S4LRU, RIGHT?

They decided to do so…

Total failure!

Why didn't it help?

# S4LRU DIDN'T REALLY WORK WELL!

It turned out that the algorithm worked well *in theory* but created a pattern of reads and writes that were badly matched to flash memory

Resulted in a whole two year project to redesign the "operating system" layer for big SSD disk arrays based on flash memory: the RIPQ project.

Once this change was made, S4LRU finally worked as hoped!

**RIPQ will not be on exams**

# RIPQ: KEY INNOVATIONS

Only write large objects to the SSD once: treats SSD like an append-only "strip of images". Same trick was needed in Haystack.

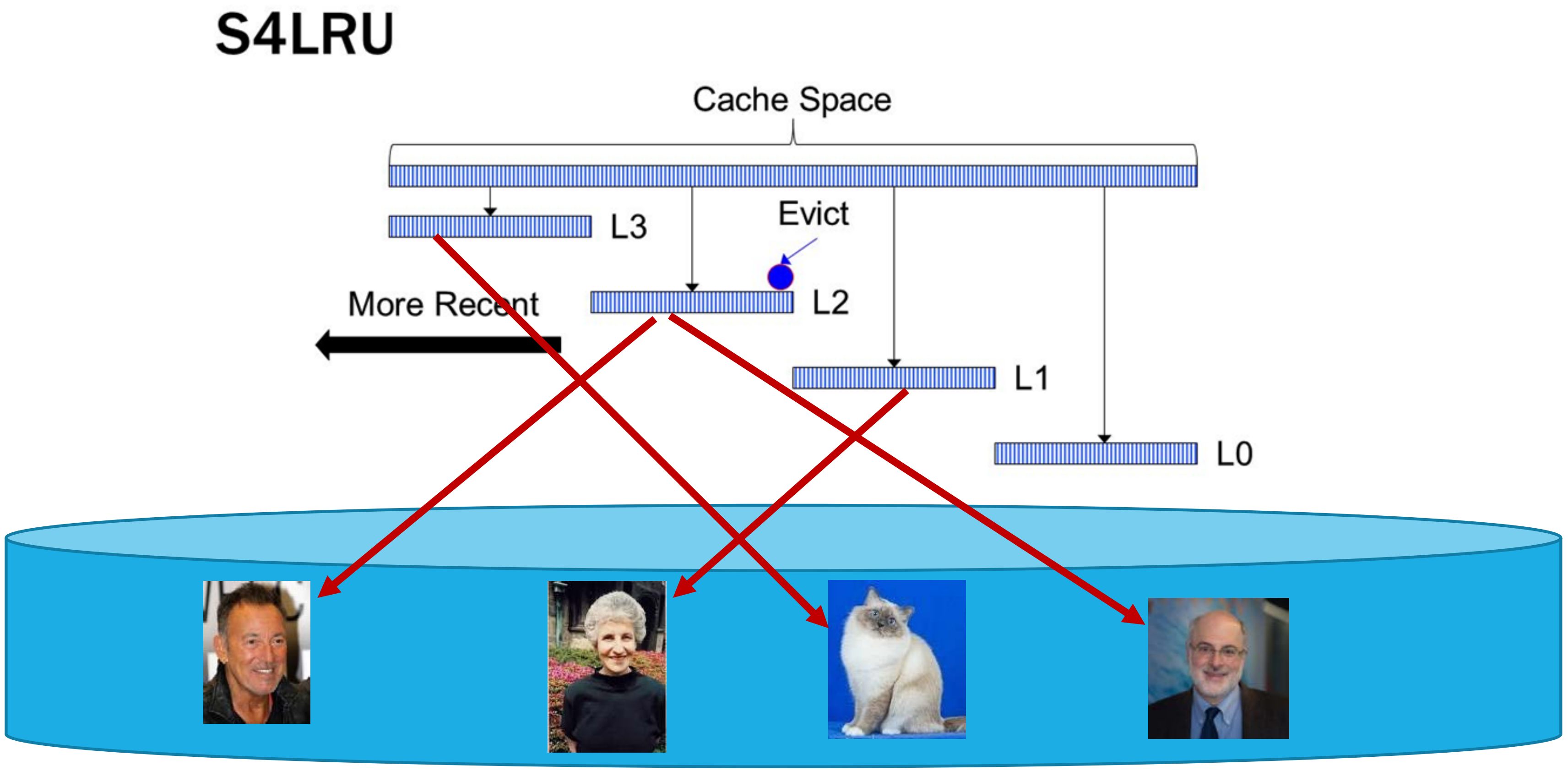
- SSD is quite good at huge sequential writes. So this model is good.

But how can they implement "priority"?

- The cache is an *in-memory data structure* with pointers onto the SSD.
- They checkpoint the whole cache periodically, enabling warm restart.

**RIPQ will not be on exams**

# RIPQ ENTRIES ARE *POINTERS* TO SSD OBJECTS




**RIPQ will not be on exams**

# ADDITIONAL OPTIMIZATIONS

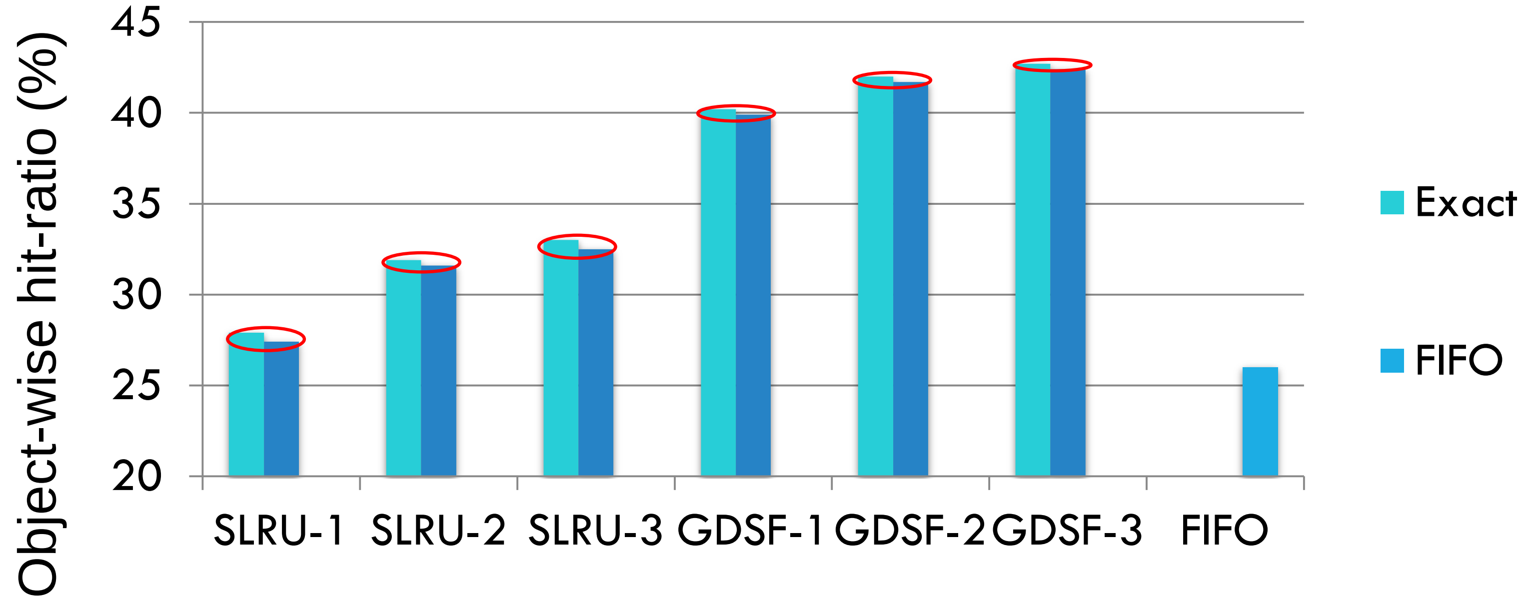
They actually store data in long “strips” to amortize the access delay across a large number of reads or writes.

And they cache hot images, plus have an especially efficient representation of the lists of pointers (for the various levels of the cache)

All of this ensures that they run the SSD in its best performance zone.

**RIPQ will not be on exams**

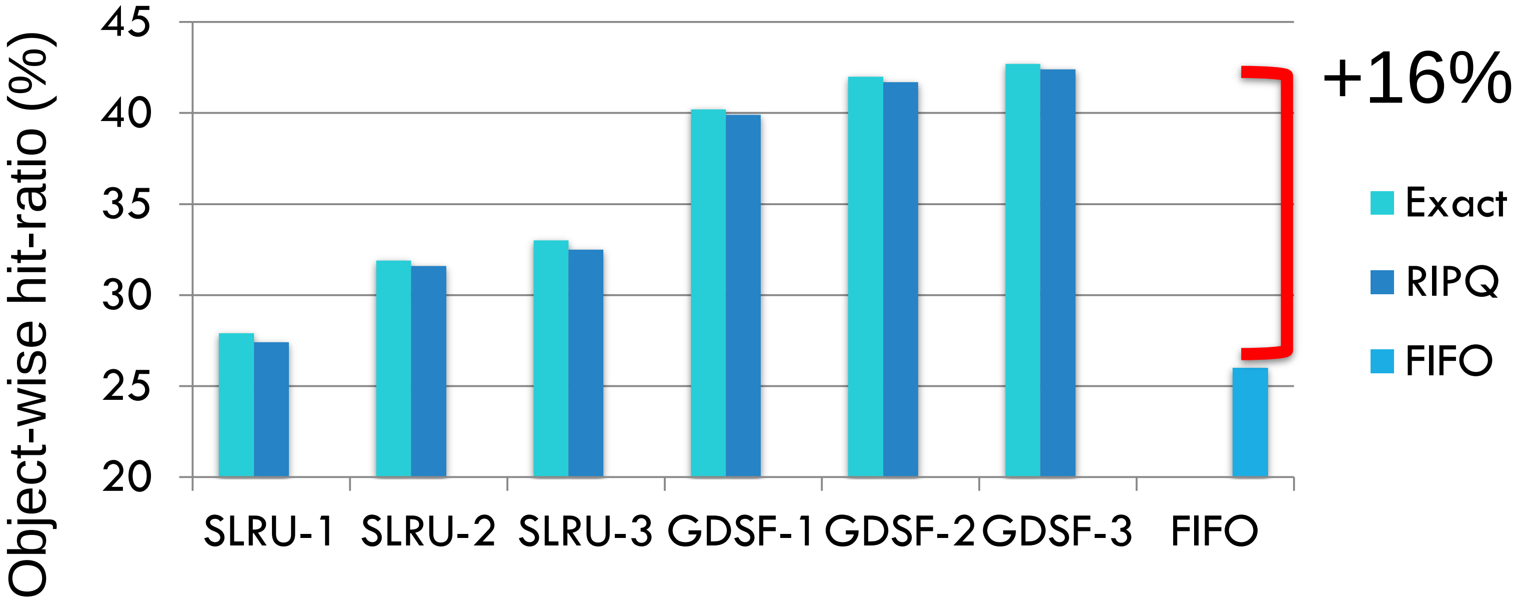
# RIPQ HAS HIGH FIDELITY




RIPQ achieves ≤0.5% difference for all algorithms

**RIPQ will not be on exams**

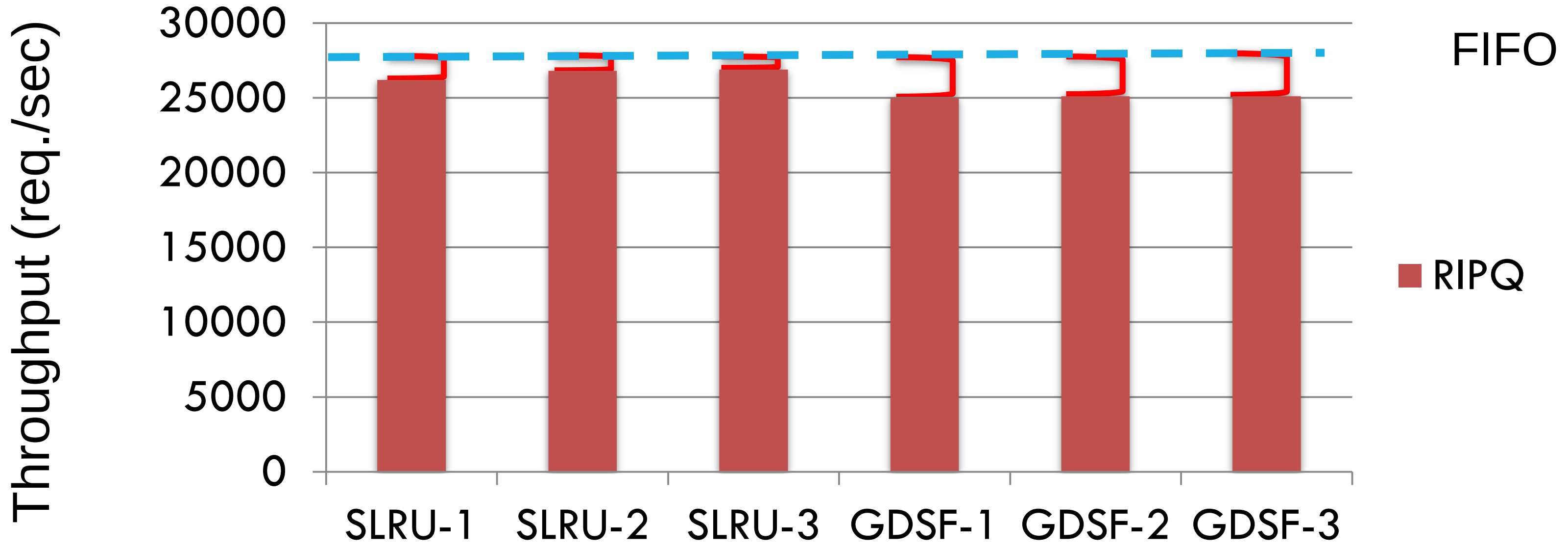
# RIPQ HAS HIGH FIDELITY




+16% hit-ratio ➔ 23% fewer backend IOs

**RIPQ will not be on exams**

# RIPQ HAS HIGH THROUGHPUT




RIPQ throughput comparable to FIFO (≤10% diff.)

**RIPQ will not be on exams**

# WHAT DID WE LEARN TODAY?

A good example of a $\mu$-service is a key-value cache, sharded by key.

The shard layout will depend on how many servers are running, and whether they are replicated. These are examples of *configuration data*.

Many $\mu$-services are designed to vary the number of servers *elastically*.

# MORE TAKE-AWAYS

Even a sharded cache poses questions about consistency.

In fact for a cache of images, CAP is a great principle.

But this doesn't make the problem trivial, it just takes us to the next level of issues.

# MORE TAKE-AWAYS

A company like Facebook wants critical μ-services to make smart use of knowledge about photo popularity, and about patterns of access.

With this information it can be intelligent about what to retain in the cache, and could even prefetch data or precompute resized versions it will probably need.

Fetching from another cache, even across wide-area links, is advantageous.

# CONCLUSIONS?

In a cloud setting, we need massive scalability. This comes from hierarchy, replication and sharding. But we also need to match solutions to the setting.

Sharing in a (key,value) model is a great way to deal with scale. But that only gets you to the next set of questions. At cloud-scale nothing is trivial!

Facebook's caching "product" is an amazing global system. Even if it was based on simple decisions, the global solution isn't simple at all!