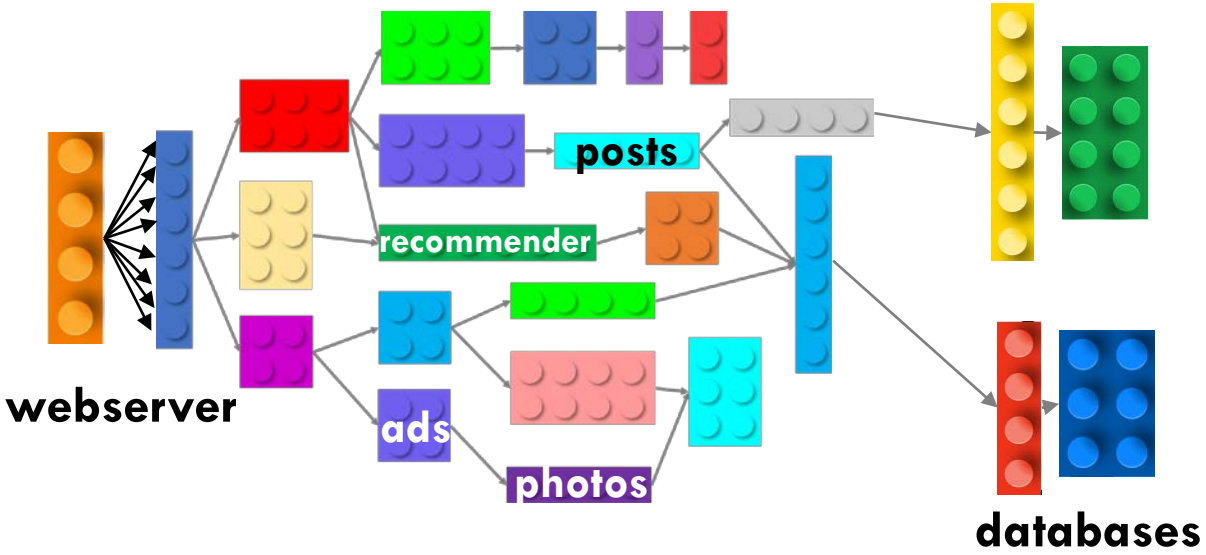
CS 5412/LECTURE 2
ELASTICITY AND SCALABILITY

Ken Birman

Fall, 2022

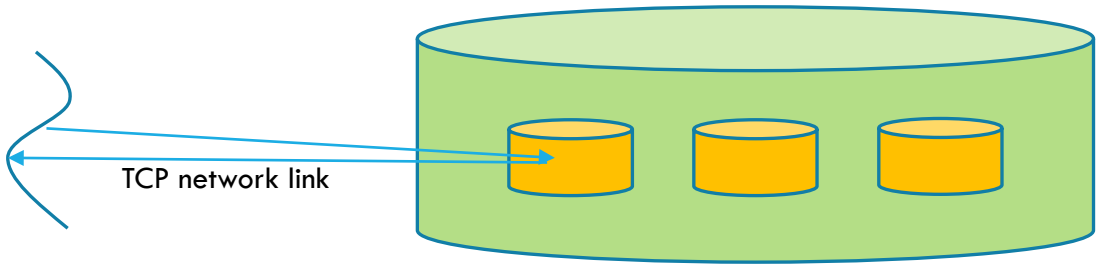
REMINDER: WHAT IS A μ-SERVICE?

The little μ is short for "micro".



Microservices

EACH μ-SERVICE…



A client program

A database implemented by 3 "instances".
This requires a special coding style

Is just some program, packaged as a "docker container"

The software was written to run with multiple side-by-side copies ("instances")

- Some services are started *on demand* and shut down when not in use. Others are always running, and often we add or remove instances as needed
- Client programs talk to service instances over the network. Looks just like a function call, but in fact a message is sent, and the reply is a message too
- Cloud services keep data in files or databases or key-value storage, updating it as needed. This makes it easy to start/stop instances

LOAD SWINGS ARE INEVITABLE

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.

CONCEPT OF ELASTIC SCALABILITY

The pools of servers making up a μ-service must support having instances added or removed *elastically*, "on demand".

We need to be sure that the performance of individual instances will remain good even if the number of instances suddenly doubles, or suddenly shrinks.

THE MAIN PROBLEM: μ-SERVICE OVERLOADS

It is very common for μ-services to need to talk to other μ-services, or from a storage server. Those shared servers can easily become hot spots.

One big thing: we could implement big services as monolithic products running on clusters. The Oracle enterprise database product works this way. But there is a limit to how big the clusters can grow.

Many smaller things: Split services into an array of "mini-servers:" **sharding.** Very easy to scale, by adding shards, but imposes limitations.

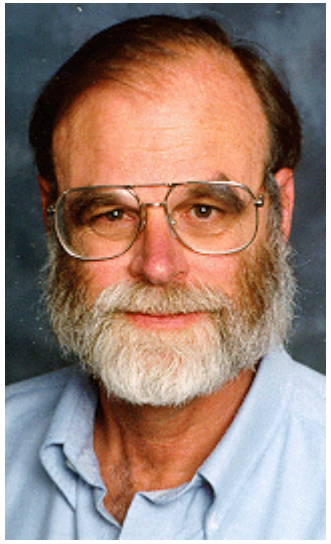
SINGLE BIG SERVICE PERFORMS POORLY… WHY?

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.

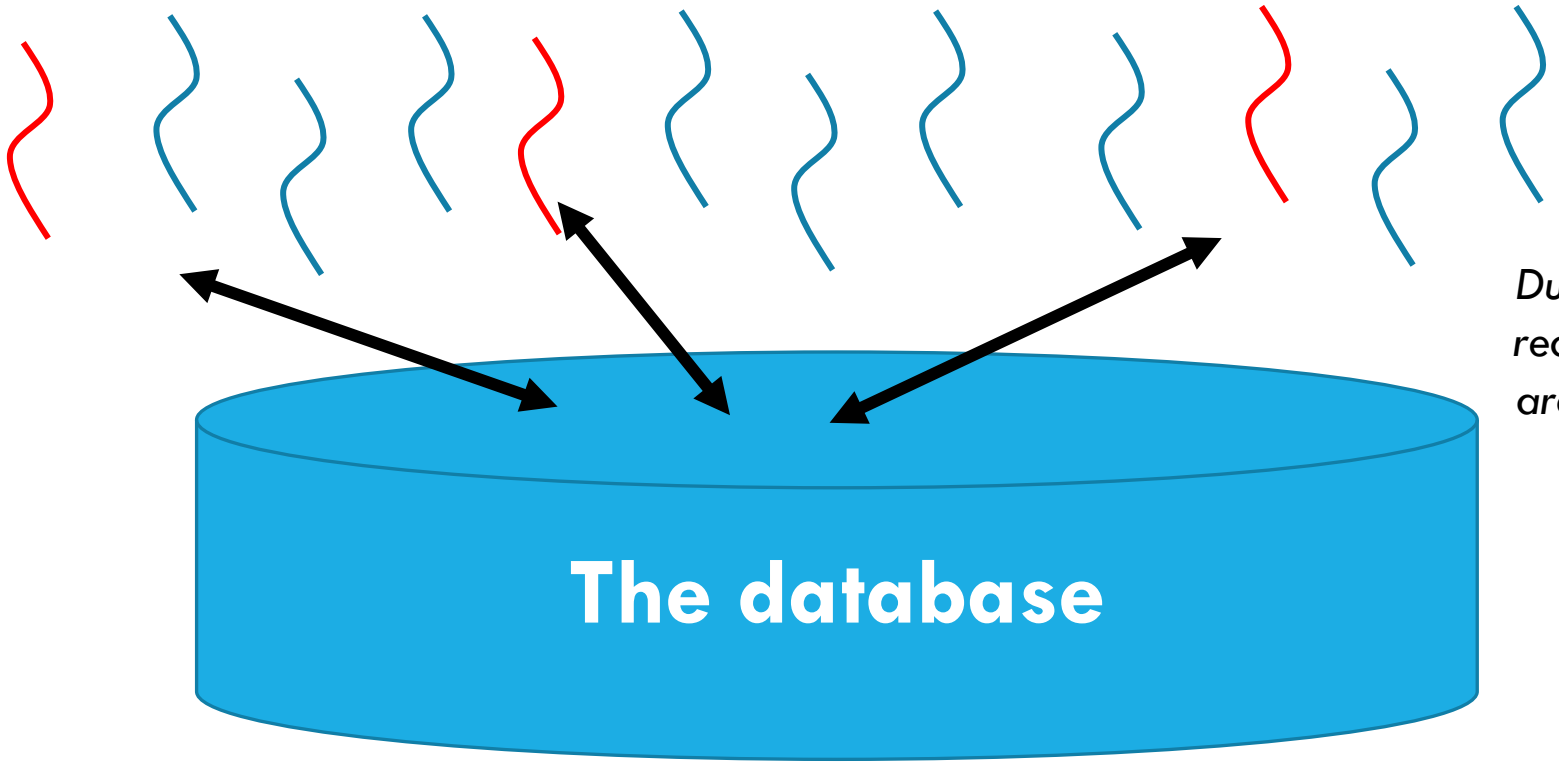
THE CORE ISSUE

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.

KEY PARAMETERS

N, the number of servers running the service.

T, the number of concurrent transactions the service is asked to perform.

Goal: For larger T, use more servers (make N bigger).

Our beta test was a success with 1000 shoppers.
But we will see as many as 100,000 at the same time.
Can we just pay for extra cloud servers?
Or must we redevelop the entire system from scratch?

GRAY'S CONCLUSION: IT WONT WORK

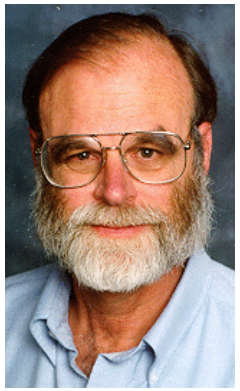
A scalable system needs to be able to handle "more T's" by adding to N

Instead, they show 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: we anticipate doubled low (T becomes 2T), so we deploy twice as many servers (N becomes 2N). But overhead rises by $2^3 2^5 = 256$x!

WHY SERVICES SLOW DOWN AT SCALE



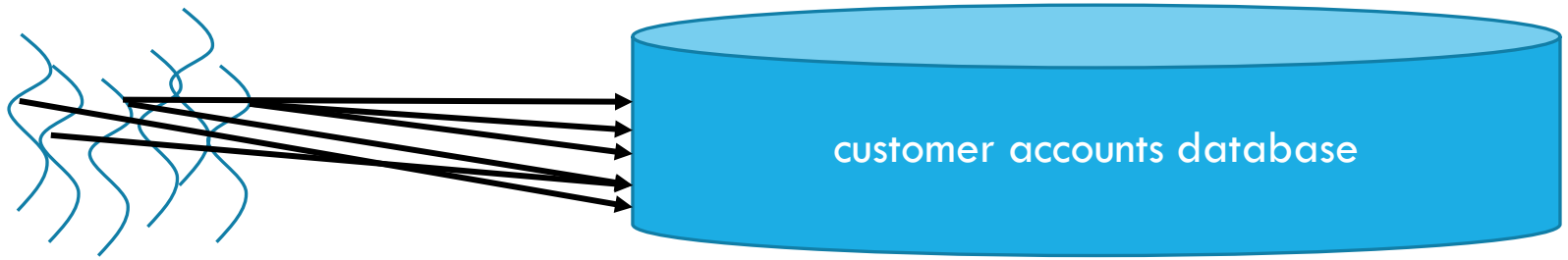
The paper pointed to several main issues:

- 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.

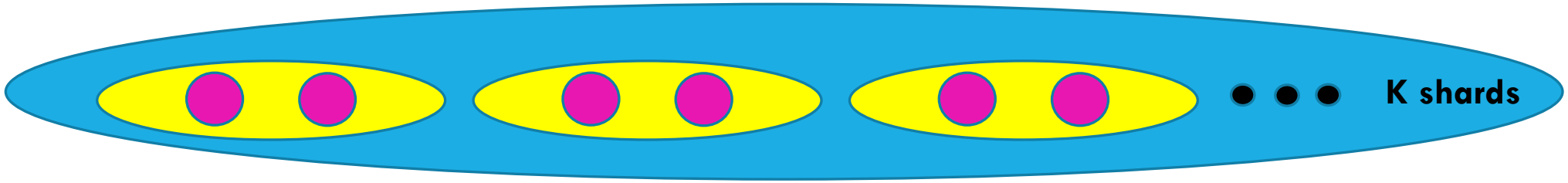
The paper actually explores multiple options for structuring the data, but ends up with similar negative conclusions *except in one specific situation*.

SHARDING: KEY IDEAS

There is one API for talking to the customer accounts database

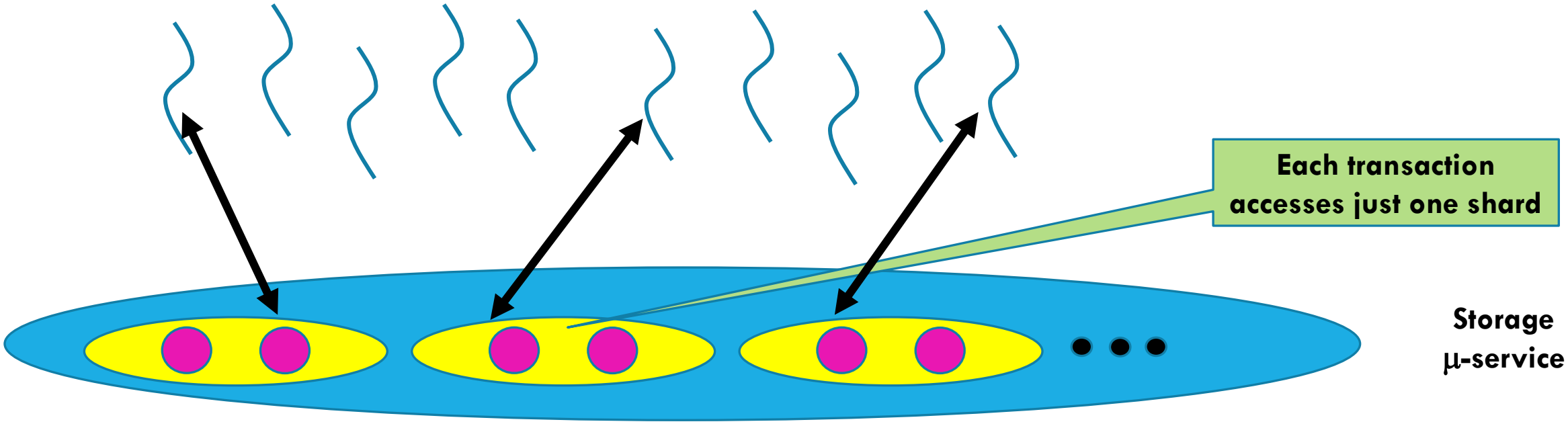


***Remote procedure calls* are one way for code in one μ-service to invoke APIs in some other μ-service**



Sharding breaks the single database into mini-databases "transparently"

SHARDS HAVE 2 (OR MORE) NODES!



The pairs of nodes here represent the idea of replication. Both of the purple nodes are individual servers, on different computers.

We use a model called *state machine replication* to keep them in sync.

BENEFITS OF REPLICATION?

Gray's formula doesn't "kick in" for small N. We actually do get a speedup, if N is small enough.

Meanwhile, we gain fault-tolerance: if some server crashes, its replica can handle requests while it recovers

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).

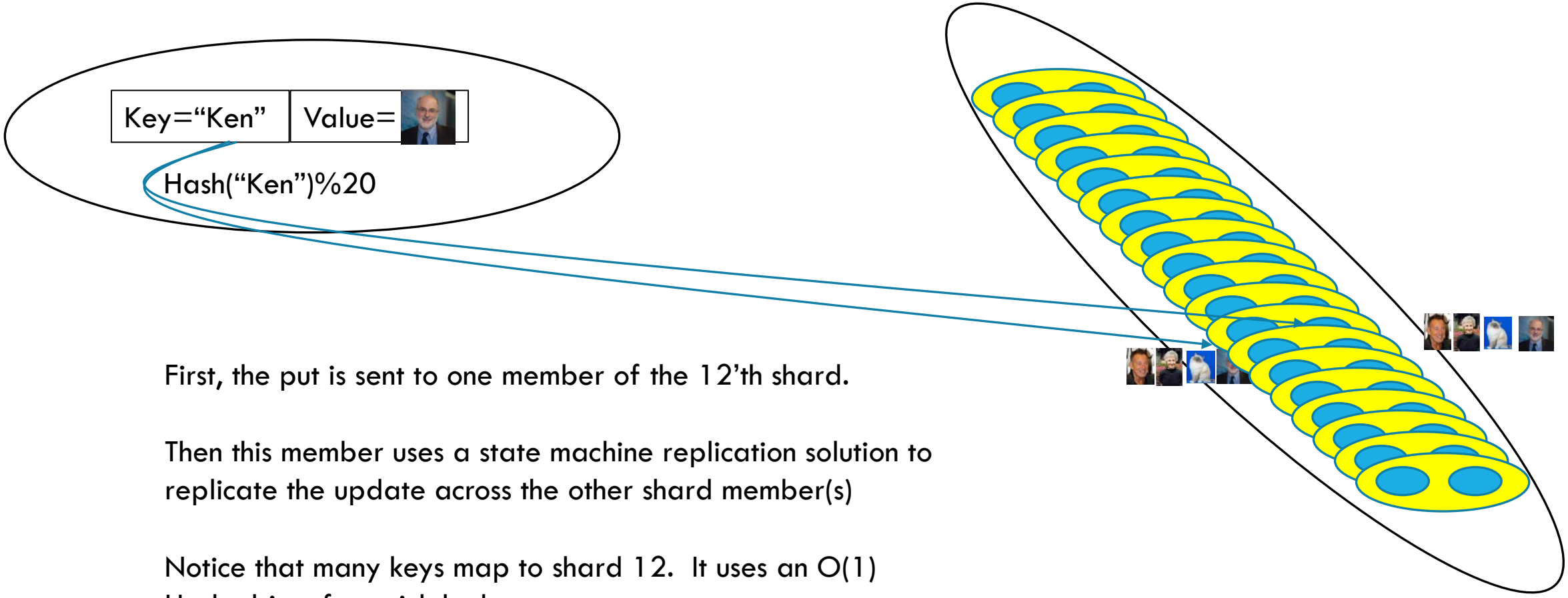
- Any server can handle a read: the code just reads the local replica.
- Replicas see the same updates in the same order, hence stay in sync.

BUT A TRANSACTION CAN ONLY TALK TO ONE SHARD AND DOES ALL ITS WORK IN "ONE STEP"

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 that needs to access multiple shards must treat the operations as concurrent, independent transactions.

PUT OPERATION



First, the put is sent to one member of the 12'th shard.

Then this member uses a state machine replication solution to replicate the update across the other shard member(s)

Notice that many keys map to shard 12. It uses an O(1) Hash object for quick lookups

SOME CHALLENGES FOR THE DEVELOPER

A key is like a "name"

Suppose your data doesn't have an obvious "key" to use. How would you create a uniq ct?

Is a data structure one objects, or many objects?

ructure, such as a table or tree, in our n one shard with one key, or spread around, using one key per object?

If the application using the data is on machine P, and the data is in a DHT on machines {Q,R,S…}, what costs would it pay?

What performance impact would these choices have?

ELASTICITY ADDS A FURTHER DIMENSION

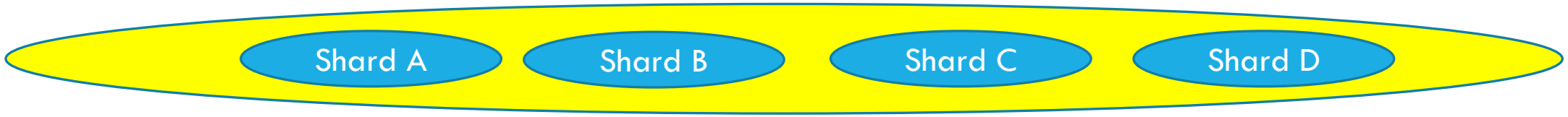
If we expect changing patterns of load, the cache may need a way to dynamically change the pattern of sharding.

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

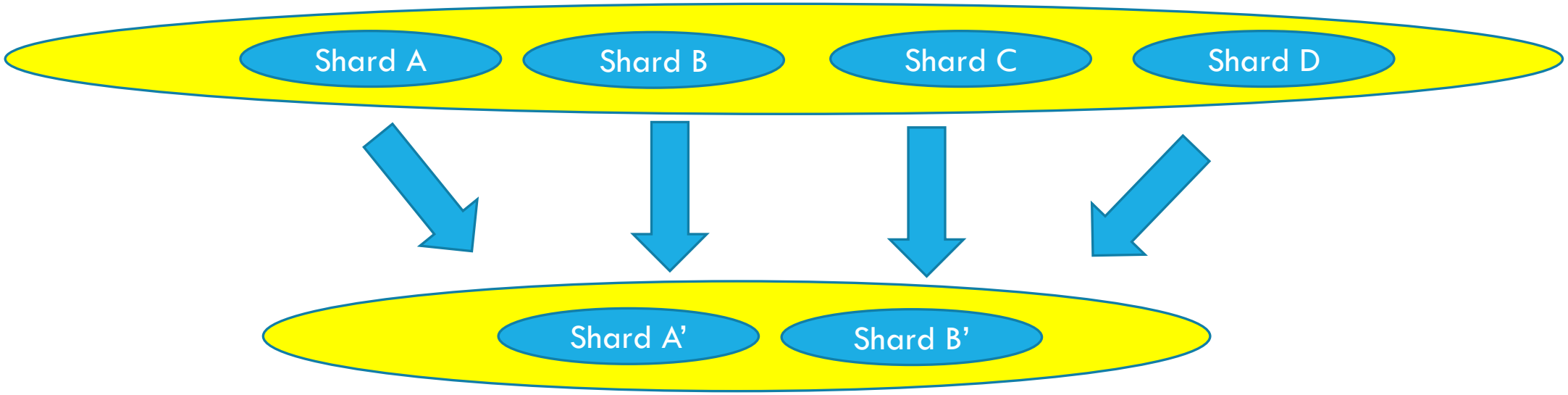
Perhaps we initially had data spread over 4 shards.



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.

FORTUNATELY, CLOUD PLATFORMS AUTOMATE THIS!

Elasticity can be hard to implement

But AWS, Amazon and Google all did the hard work for us.

When we use CosmosDB, it automatically adjusts based on load, and the user isn't even aware of it!

EXAMPLE SHARDED 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 (CosmosDB, Cassandra, Dynamo DB, MemCached, DB Rocks) 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: FACEBOOK'S CONTENT-SERVING INFRASTRUCTURE

We are all aware of how amazingly fast Facebook can be

Internally, Facebook's software for content serving is a heavily sharded service built with lots of μ-services that talk to each other.

The "transactions" are all very simple reads and writes. So they can be done in a single operation on a single shard at a time

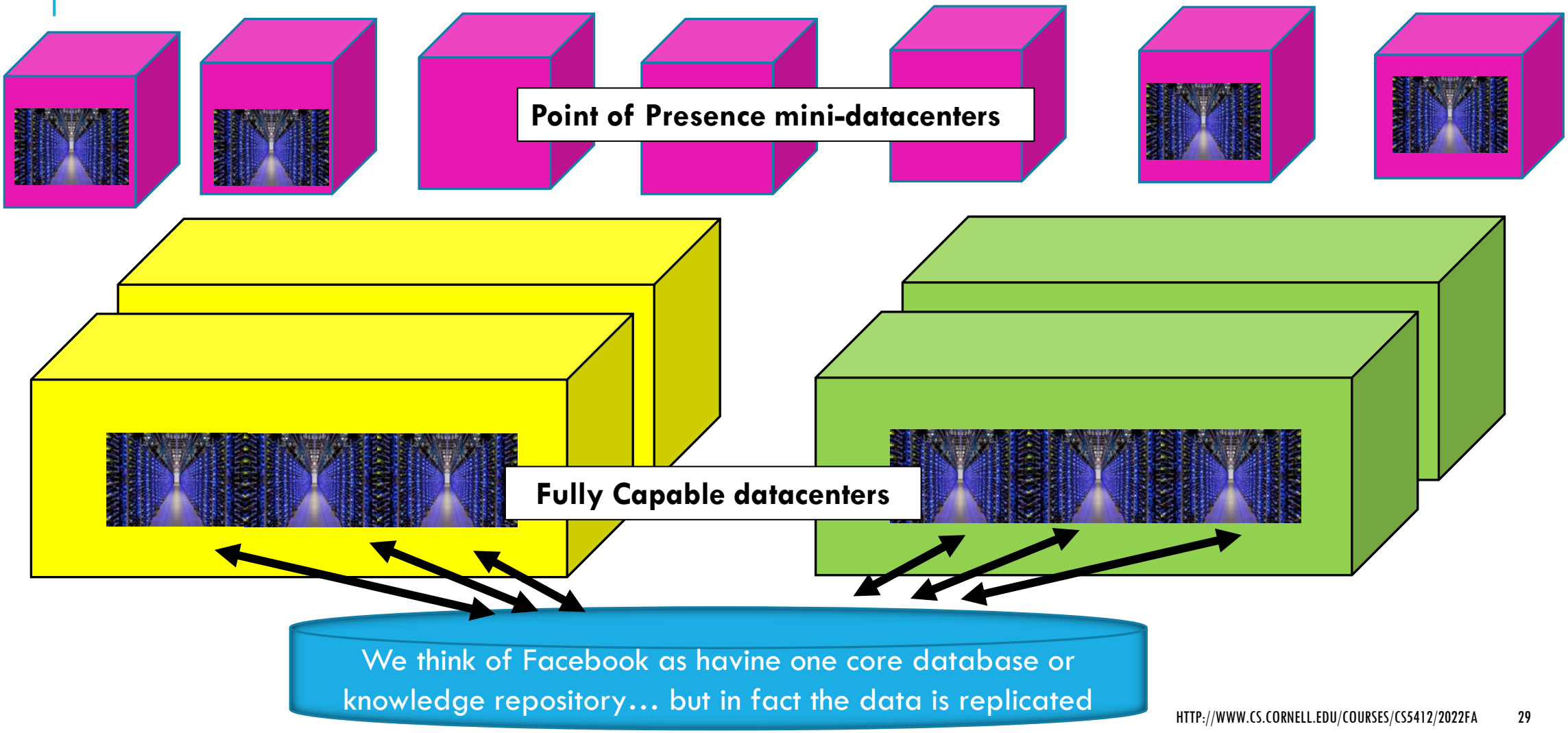
USE CASE: FB CONTENT DELIVERY NETWORK

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…



CONTENT CACHE GOALS

Support very fast lookup by any client from anywhere in the world

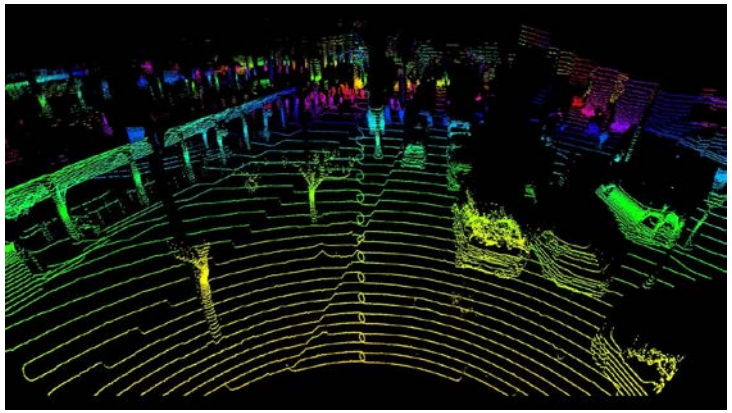
Minimize the number of requests that reach the database backend

Uses sharded KVS at every level!

… 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?

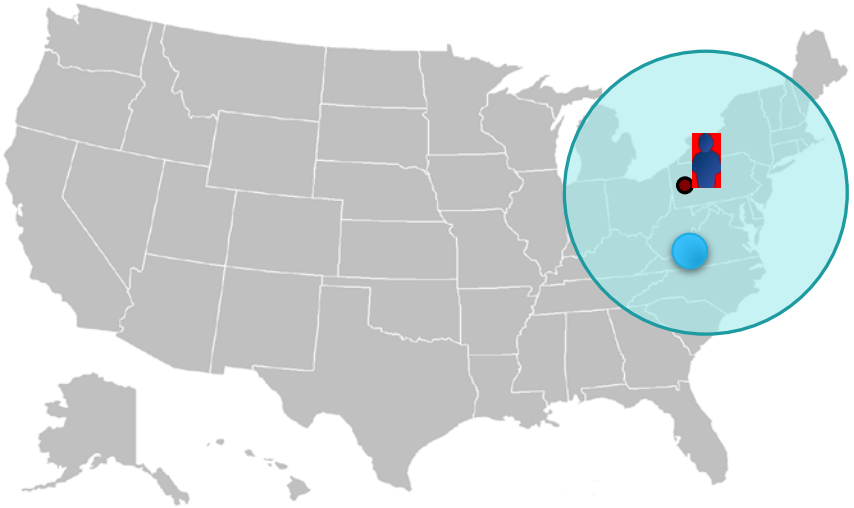
Goal: a μ-service we can run anywhere, and that can adapt in size as loads vary.

The keys would be photo or video id's.

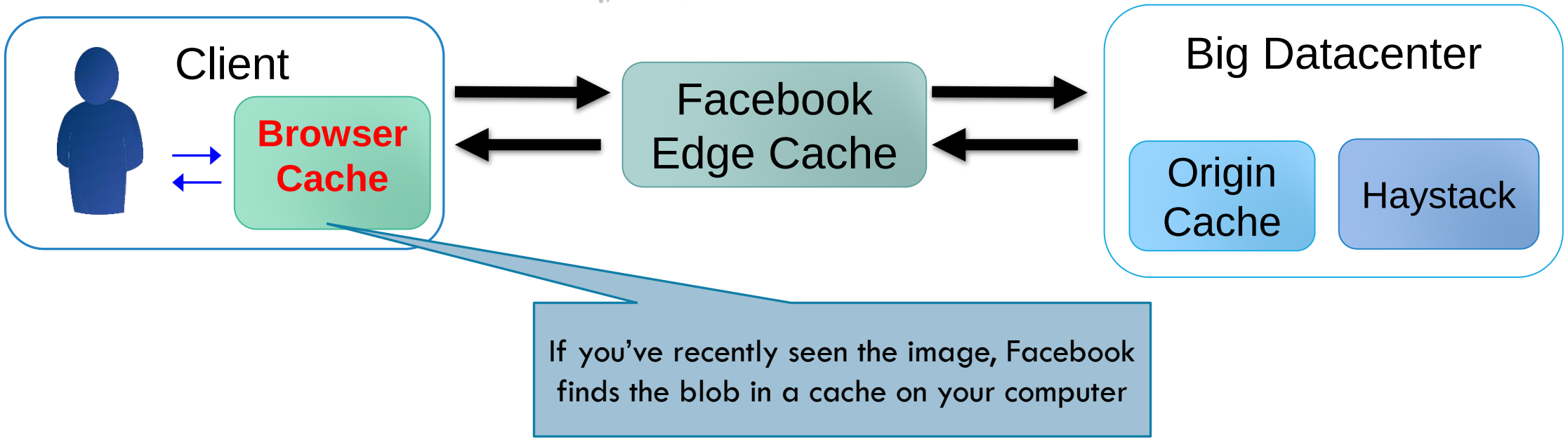
For each unique blob, FB has a special kind of tag telling us the resized dimensions of the particular instance we are looking at.

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

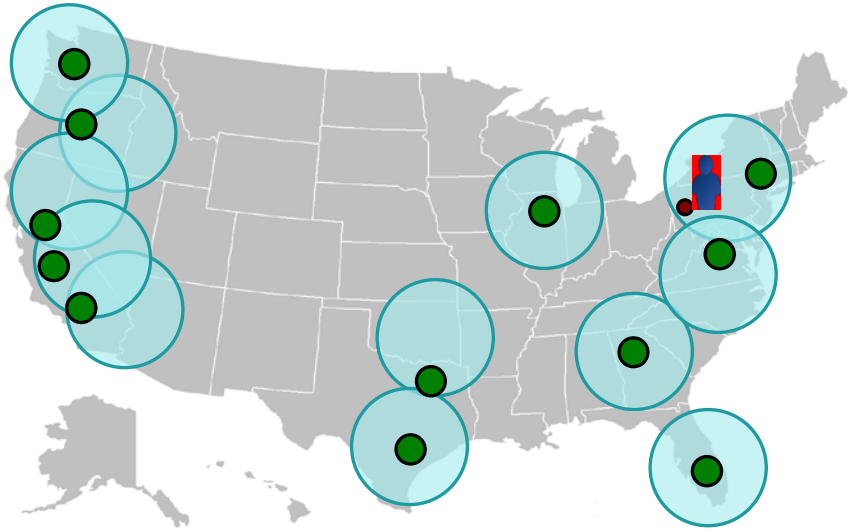
True data center holds the original photo in Haystack

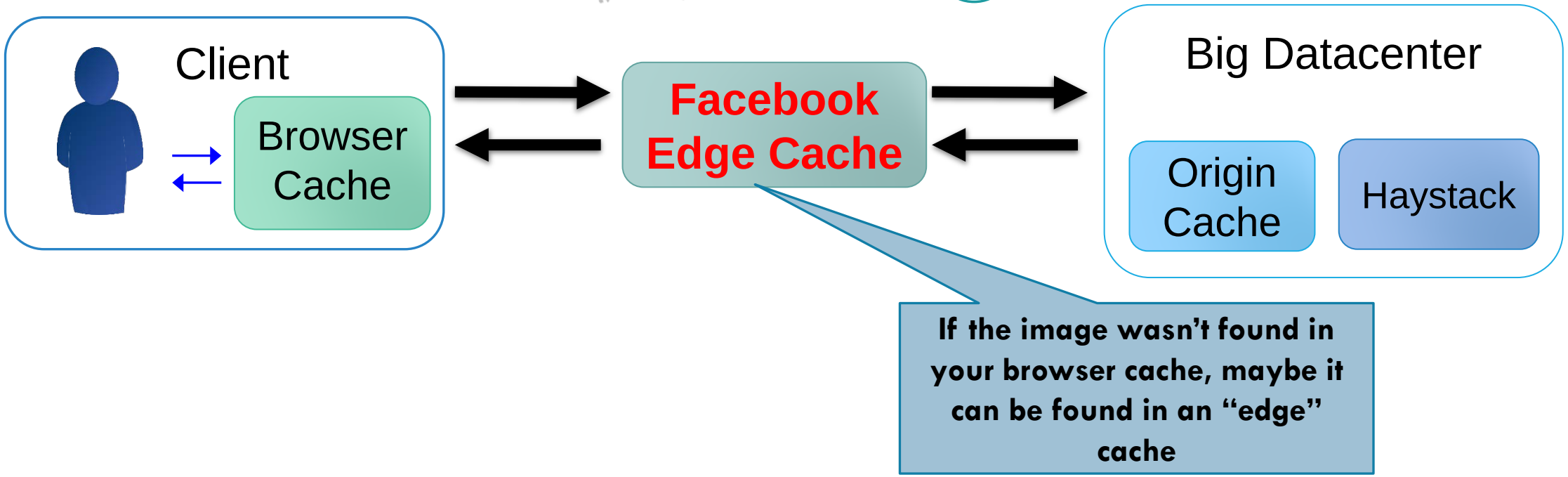


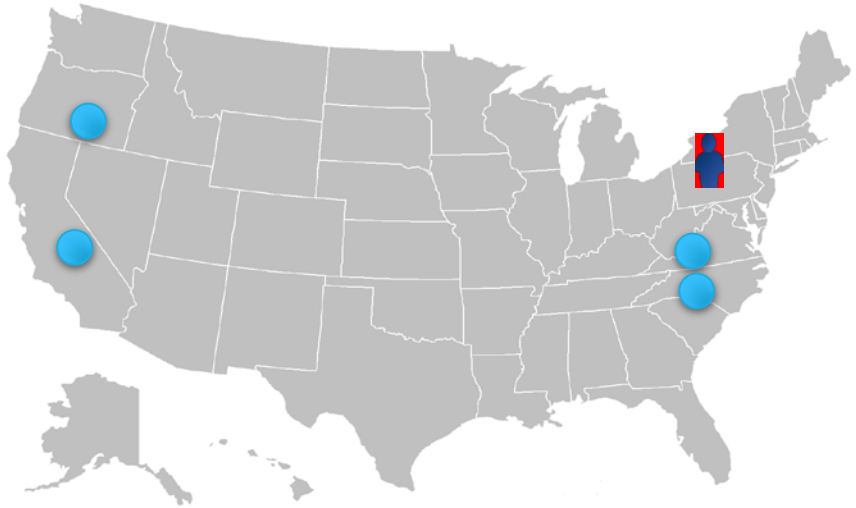
First, FB fetches your feed. This will have URLs with the image ID embedded in them. The browser tells the system what size of screen it has.



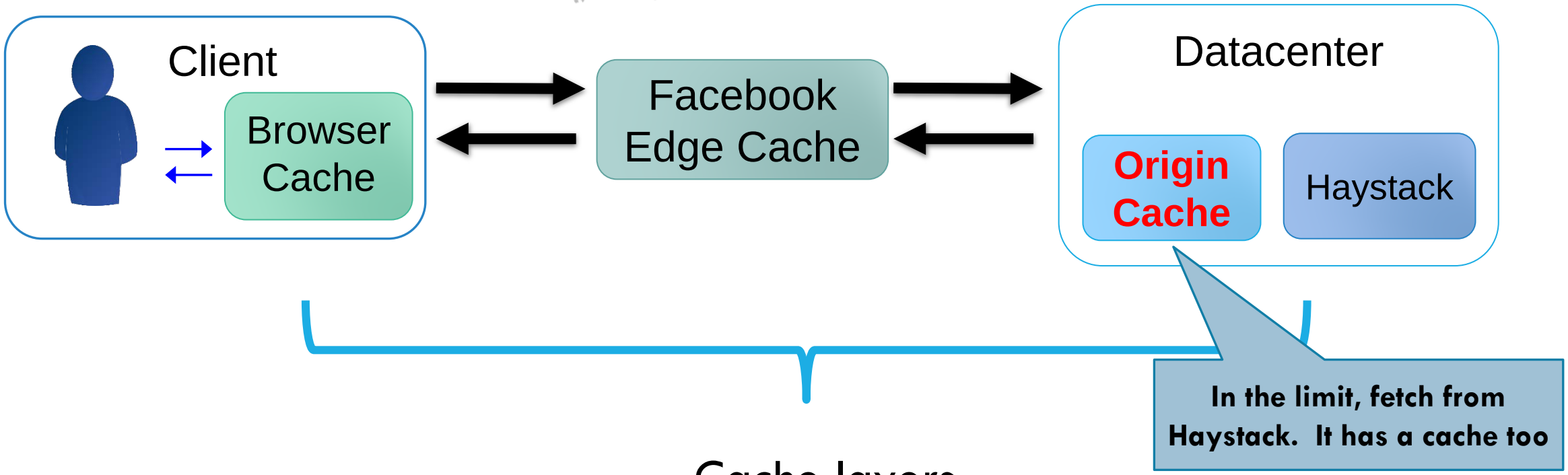
With a UID and a target size, we can search for the blob in the nearest point of presence cache.



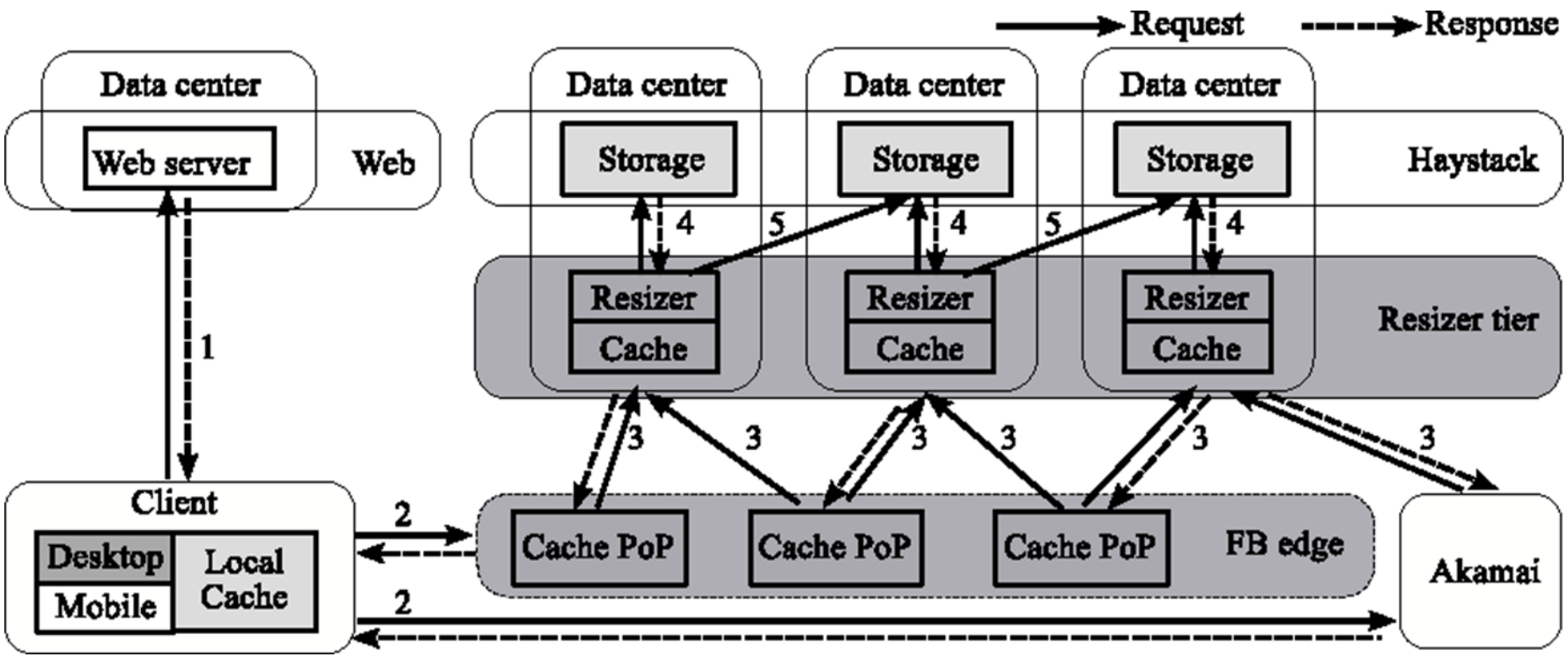




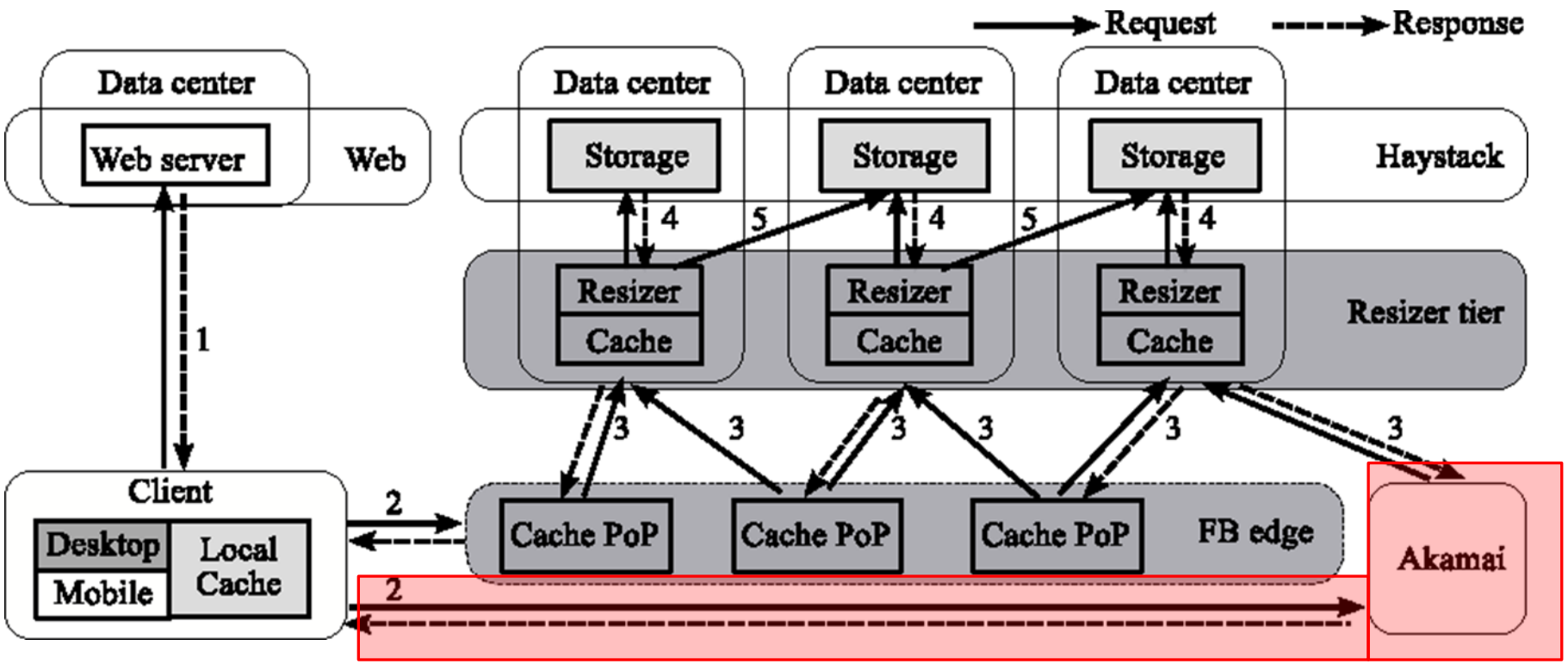
Origin:
Coordinated
FIFO
Main goal:
traffic sheltering



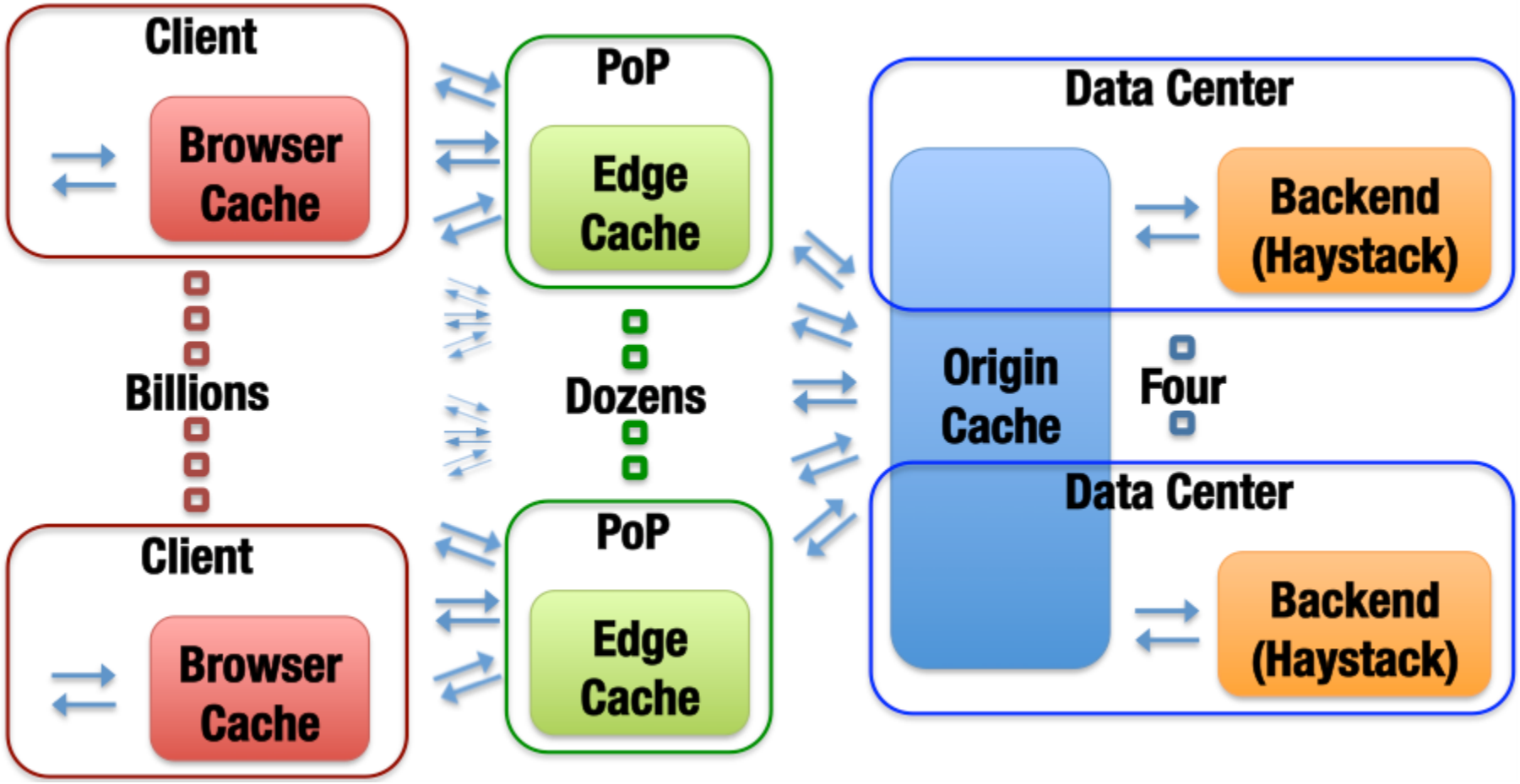
ARCHITECTURAL DETAIL (COMPLEX)
DARK GRAY: WE INSTRUMENTED IT
PALE GRAY: WE CAN FIGURE OUT ITS BEHAVIOR



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

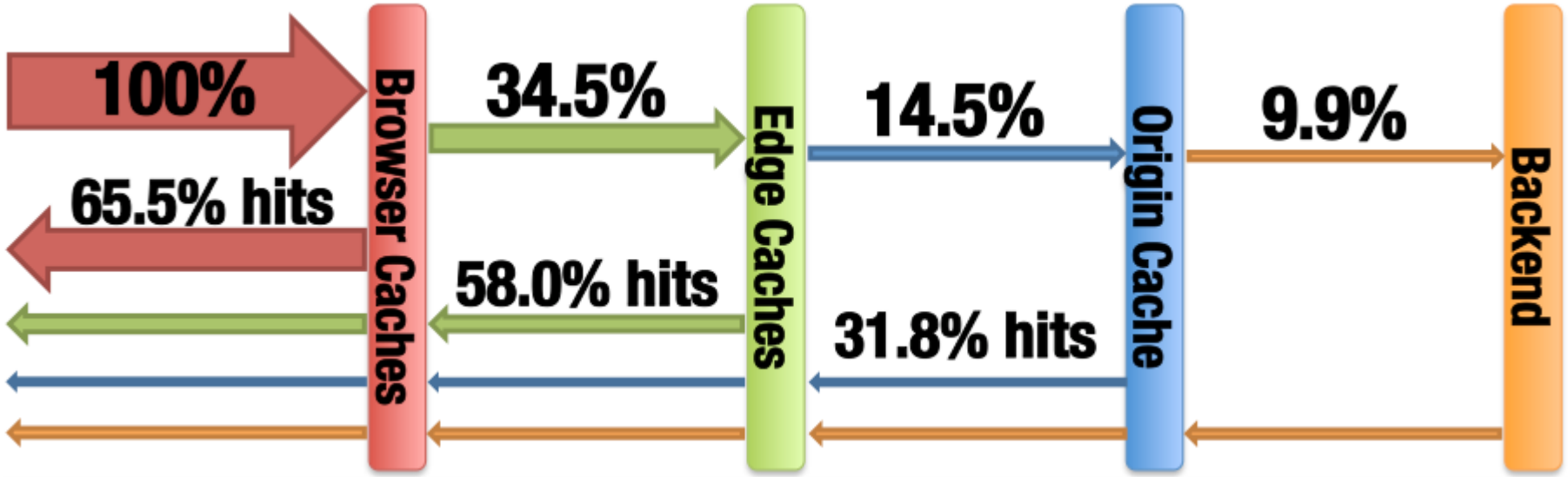


BLOB-SERVING STACK (THE FB PORTION)



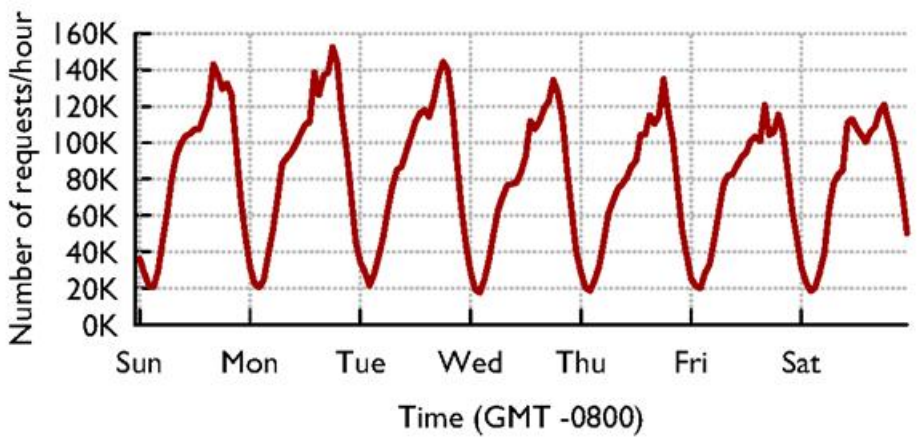
WHAT WE OBSERVED

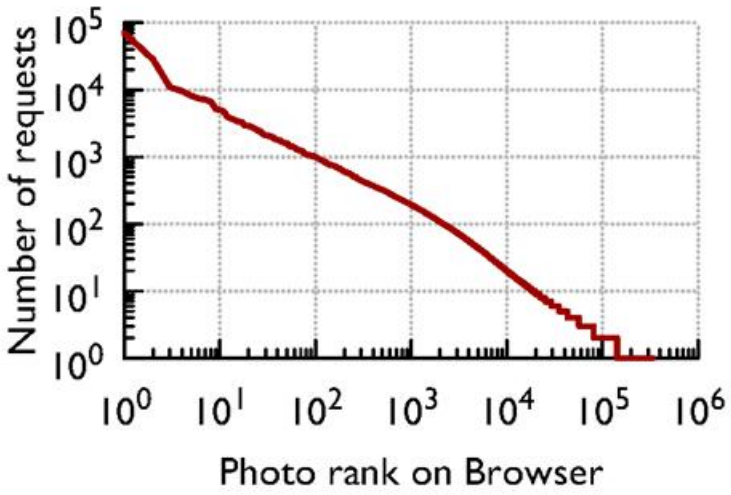
Month long trace of photo accesses, which we sampled and anonymized. Captures cache hits and misses at every level.



CACHES SEE "CIRCADIAN" PATTERNS

Accesses vary by time of day…



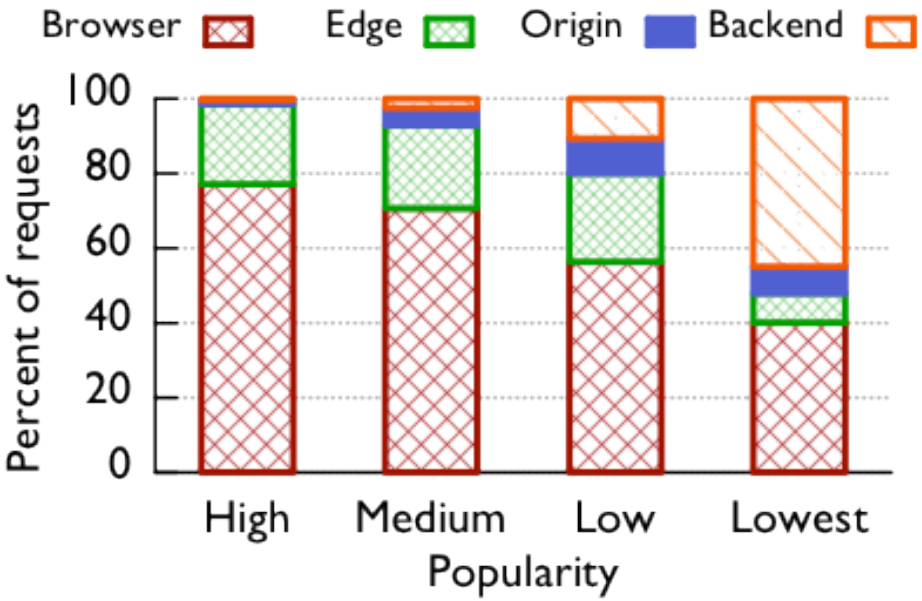


… and by photo: Some are far more popular than others

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



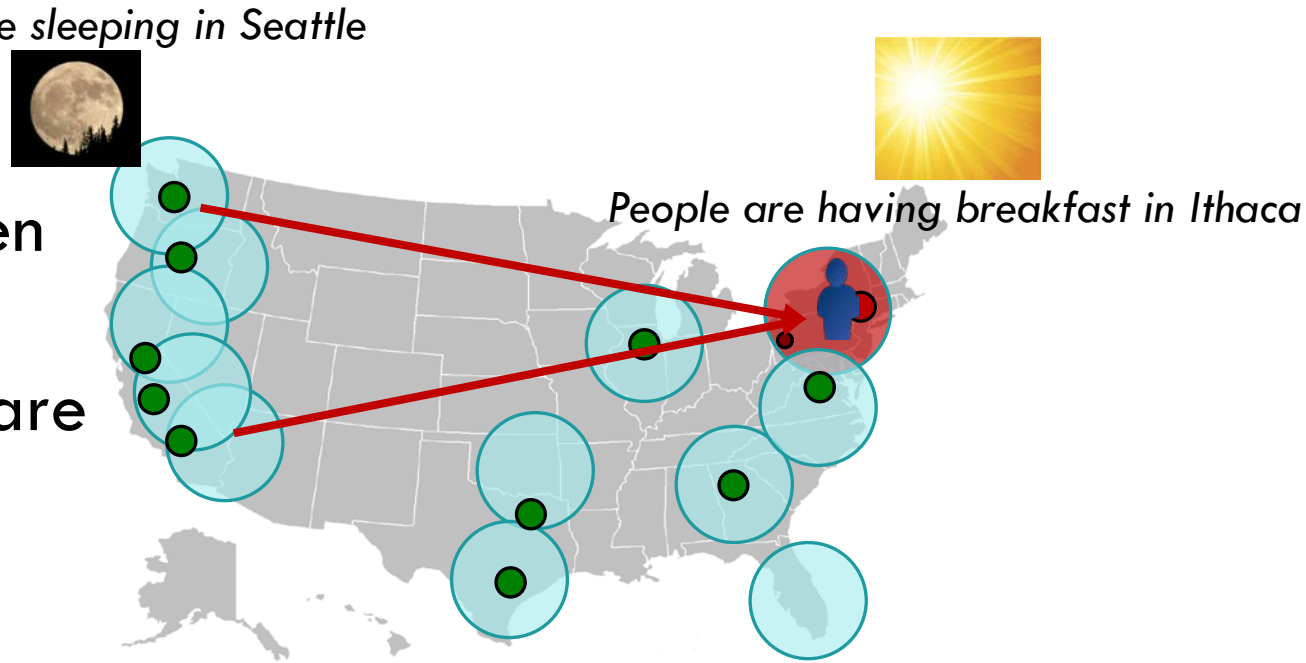
GEO-SCALE CACHING

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)

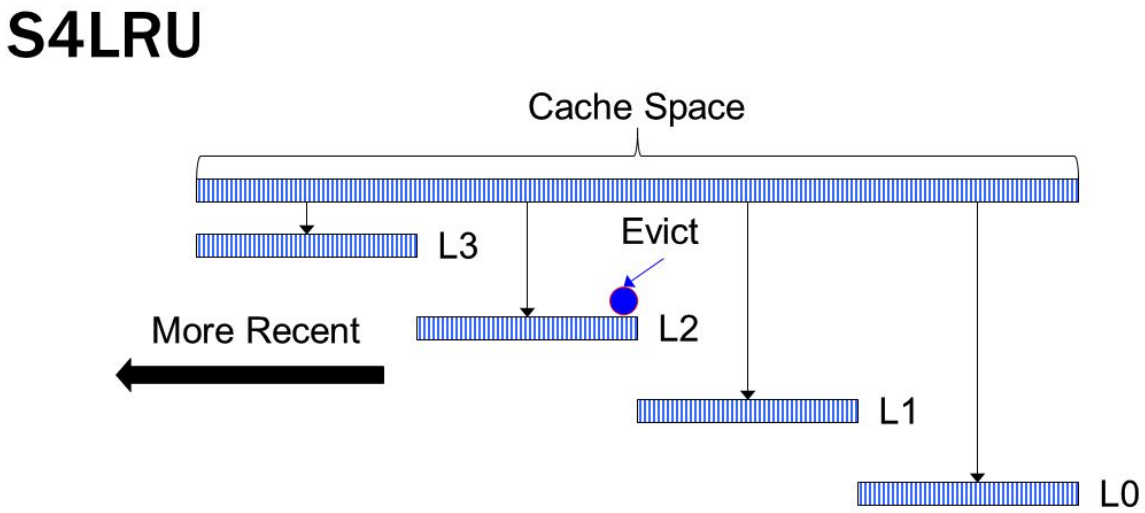
But if Boston doesn't have it or is overloaded, they casually reach out to places like Spokane, or Arizona, especially during periods when those are lightly loaded, like 5am PT!

CACHE RETENTION/EVICTION POLICY

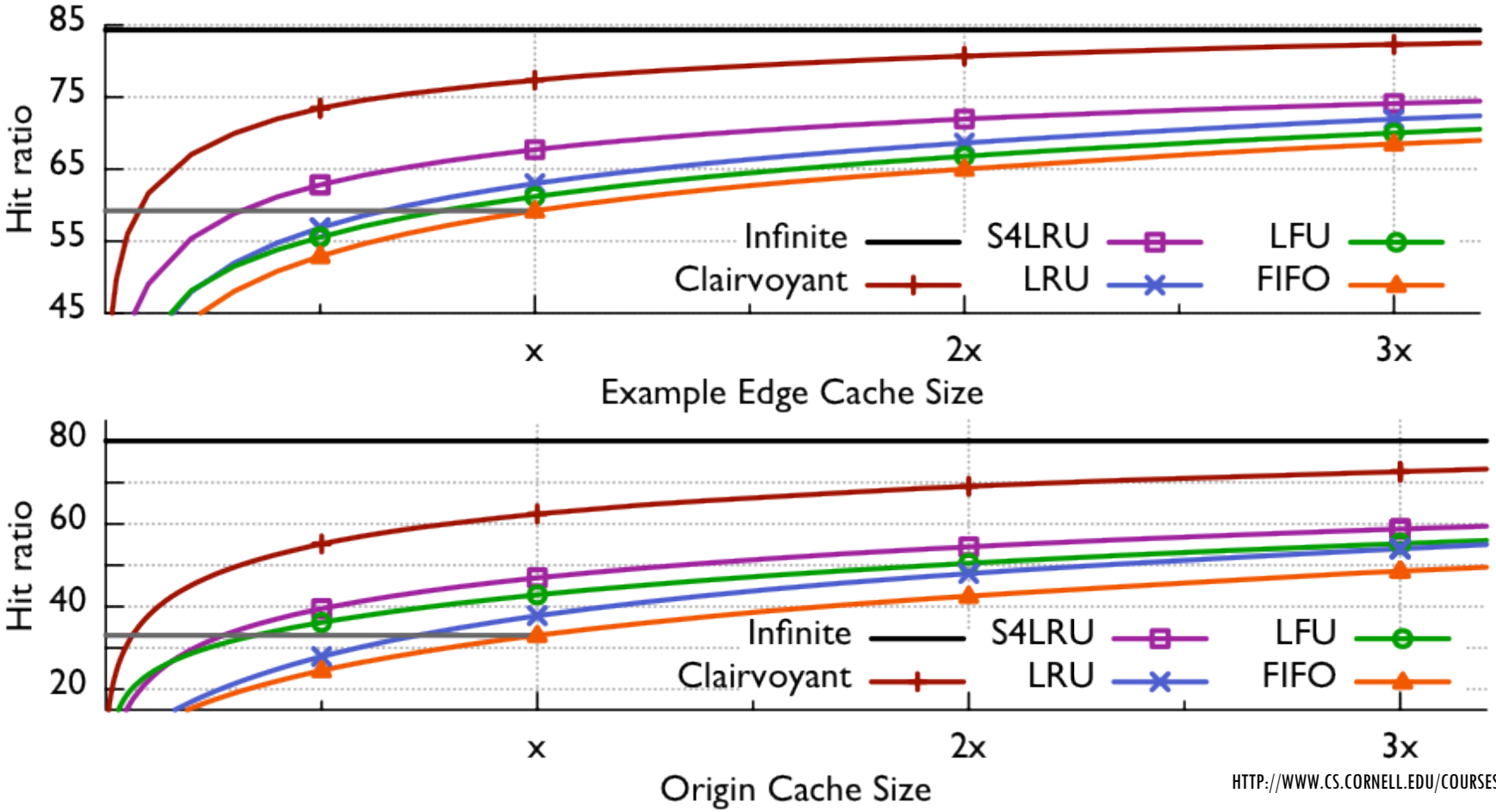
Facebook was using an LRU policy.

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.

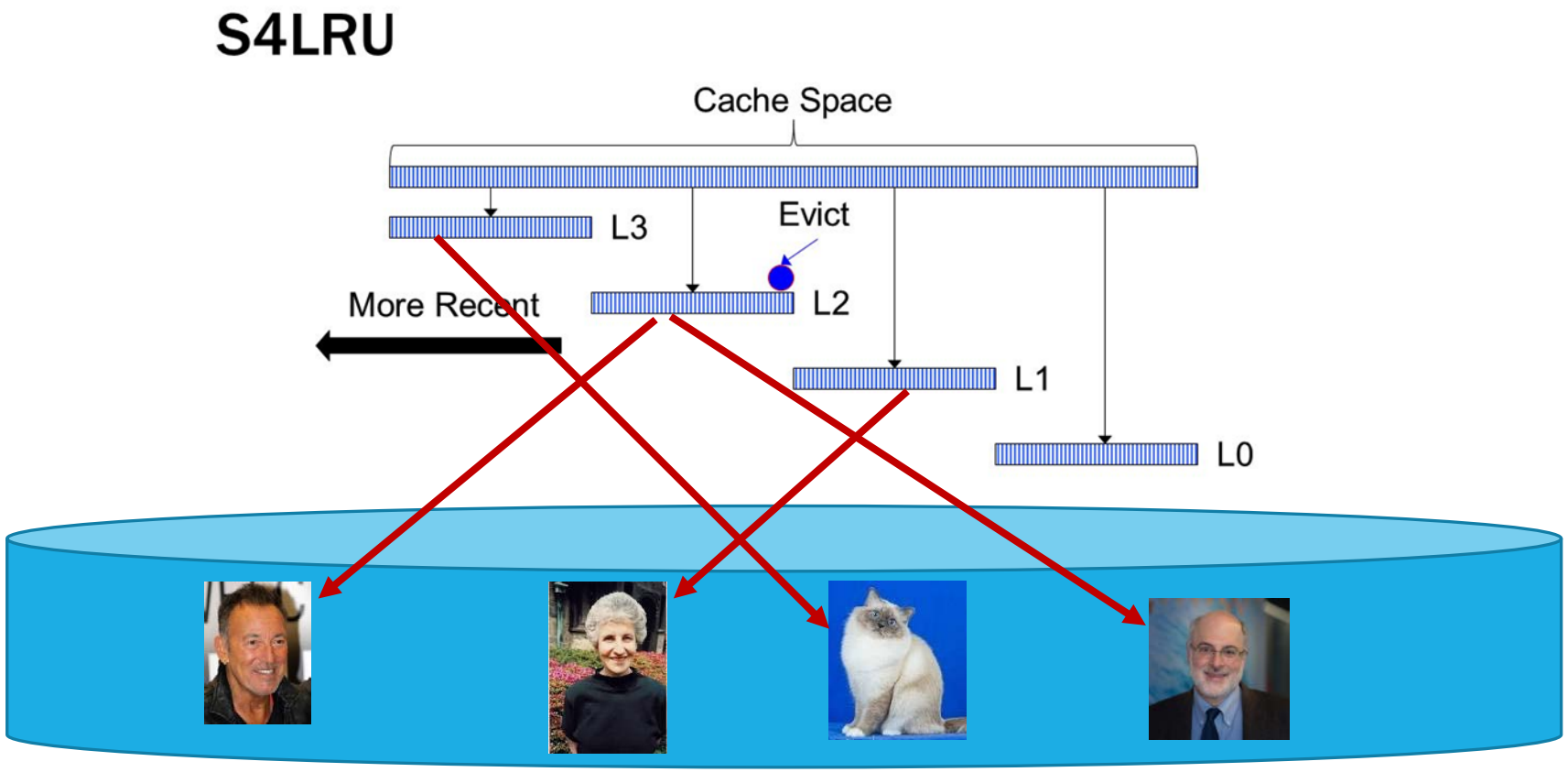
- SSD is quite good at huge sequential writes. So this model is good.

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

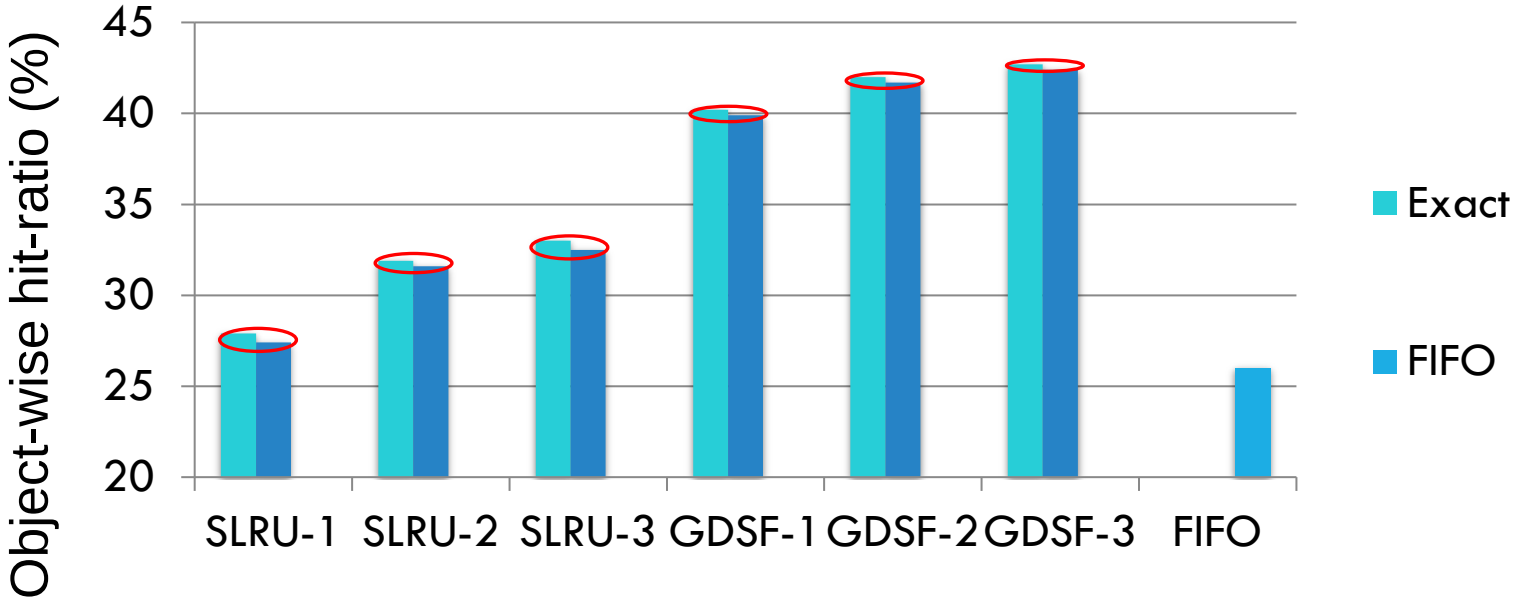
They actually store data in long "strips" to amortize the access delay across a large number of reads or writes.

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

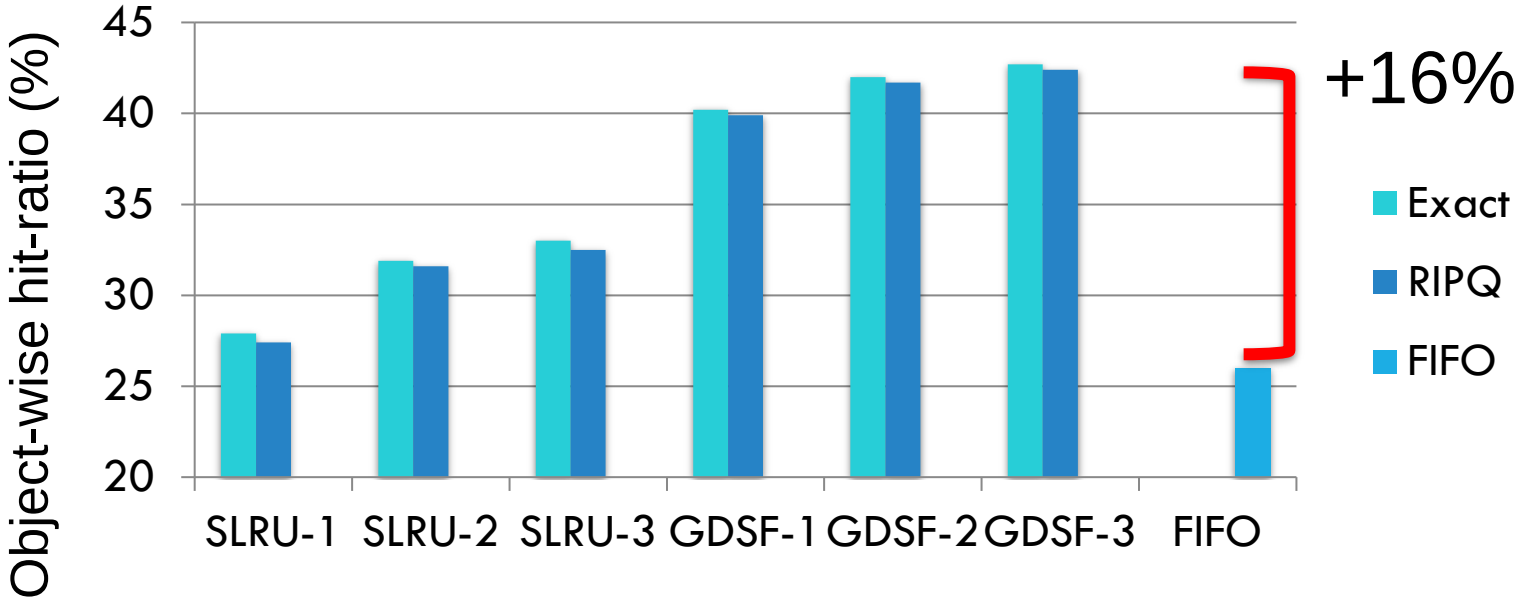
RIPQ HAS HIGH FIDELITY



RIPQ achieves ≤0.5% difference for all algorithms

RIPQ will not be on exams

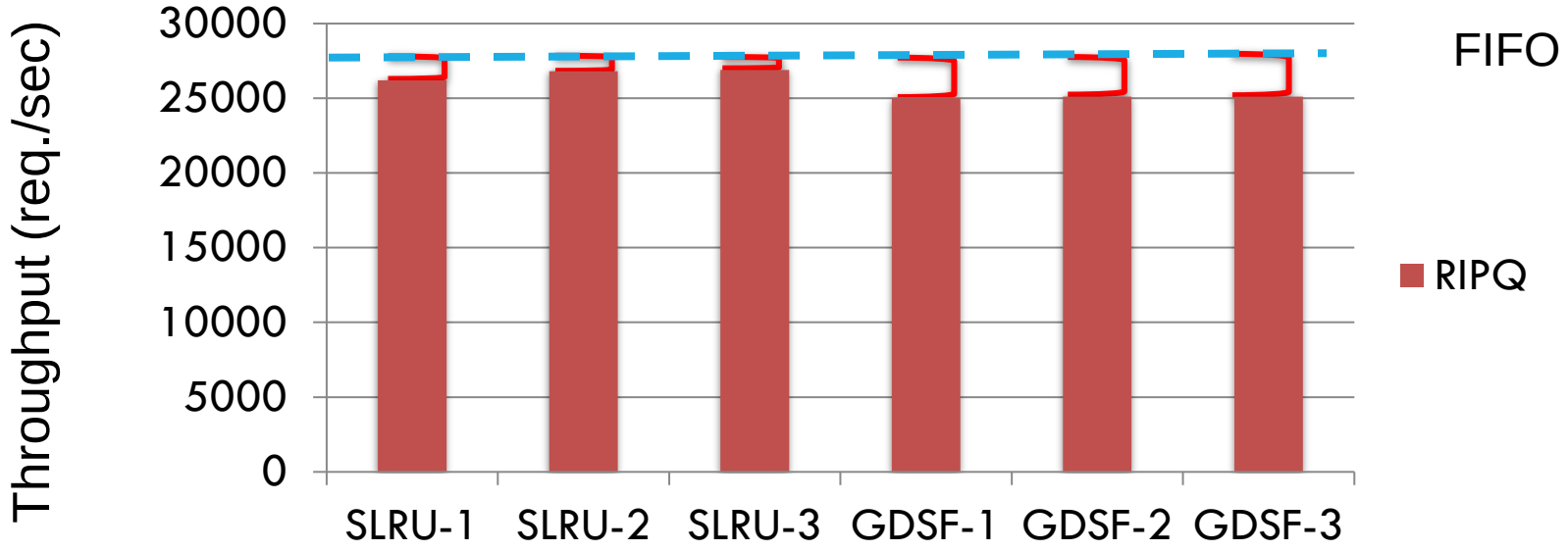
RIPQ HAS HIGH FIDELITY



+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 μ-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 μ-services are designed to vary the number of servers *elastically*.

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!