Introduction to Designing Modern Web-Scale Applications

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

These slides are lightly modified versions of slides from Prof. Ken Birman's course on Cloud Computing

How did Today's Cloud Evolve?

- Prior to ~2005, data centers claimed to be designed for high scalability and availability
  - Amazon had especially large ones, to serve its web requests
  - The real goal was just to support online shopping

- Their system wasn't very reliable
  - Core problem was scaling
  - Everything ran slowly
  - Amazon's computers were overloaded and often crashed

Amazon Experiment

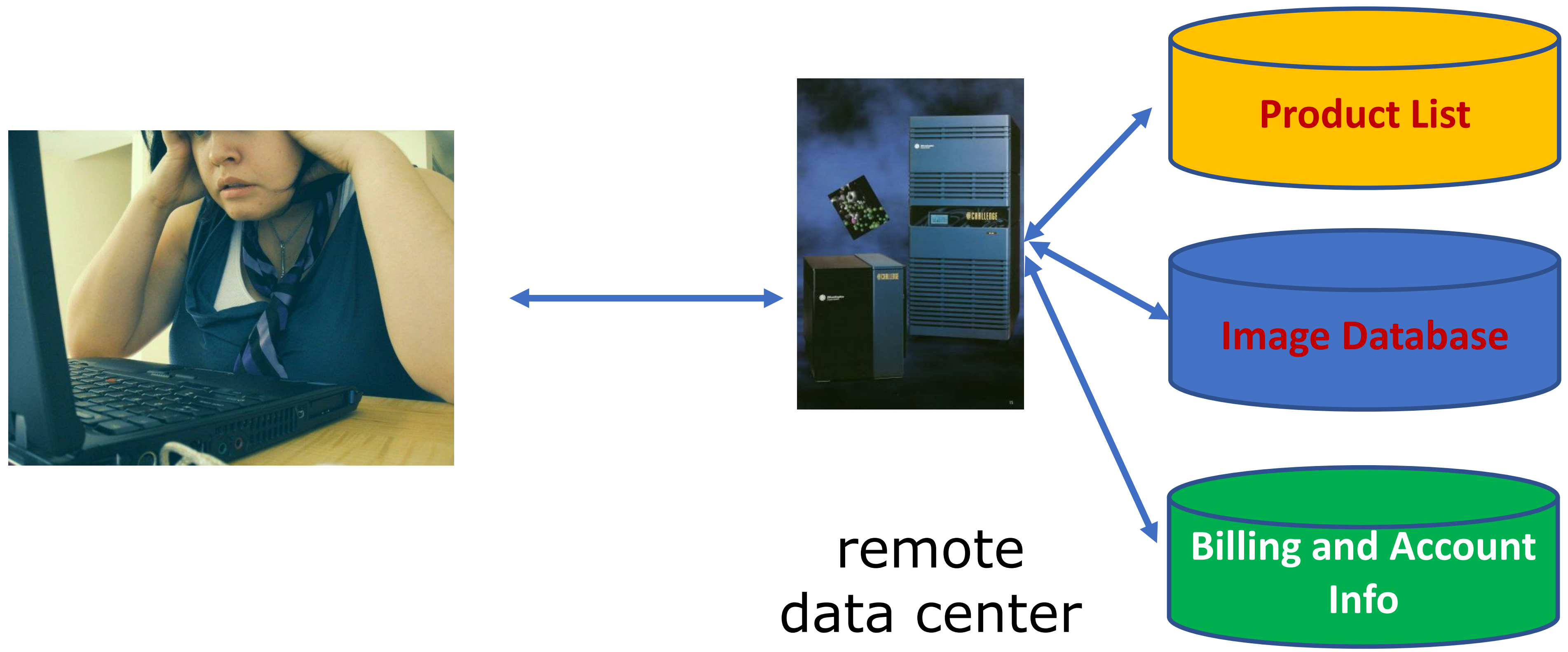
A sprint to render your web page!

- At Amazon, they tried an "alpha/beta" experiment
  - When web page was rendered fast (< 100ms), customers were happy
  - For every 100ms delay, purchase rates dropped 1%
- Conclusion: speed at scale determines revenue
  - And revenue shapes technology
  - An arms race to speed up the cloud

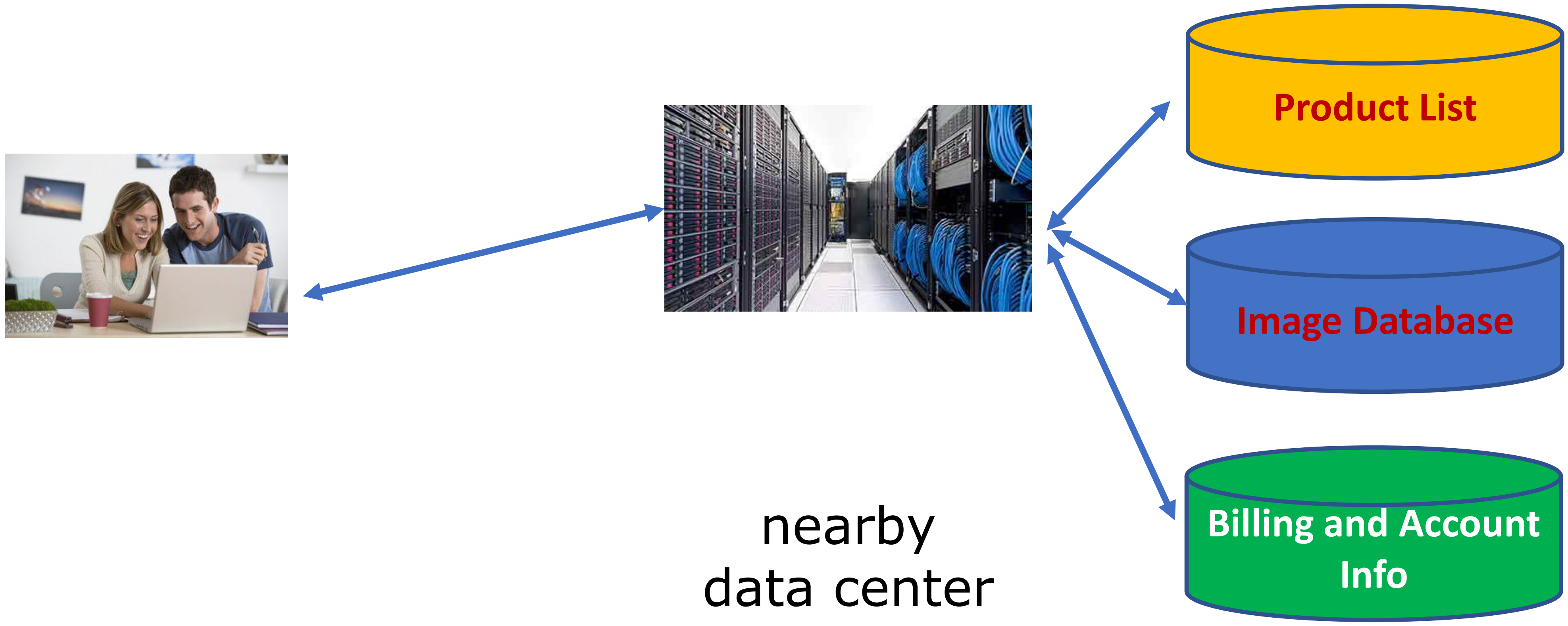
Starting around 2006, Amazon led in Reinventing Data Center Computing

- Amazon reorganized their whole approach
  - Requests arrived at a “first tier” of lightweight servers
  - These dispatched work requests on a message bus or queue
  - The requests were selected by “microservices”, executing in parallel using elastic pools
  - One web request might involve tens or hundreds of microservices!
- They also began to guess your next action and precompute what they would need to answer your next query or link click

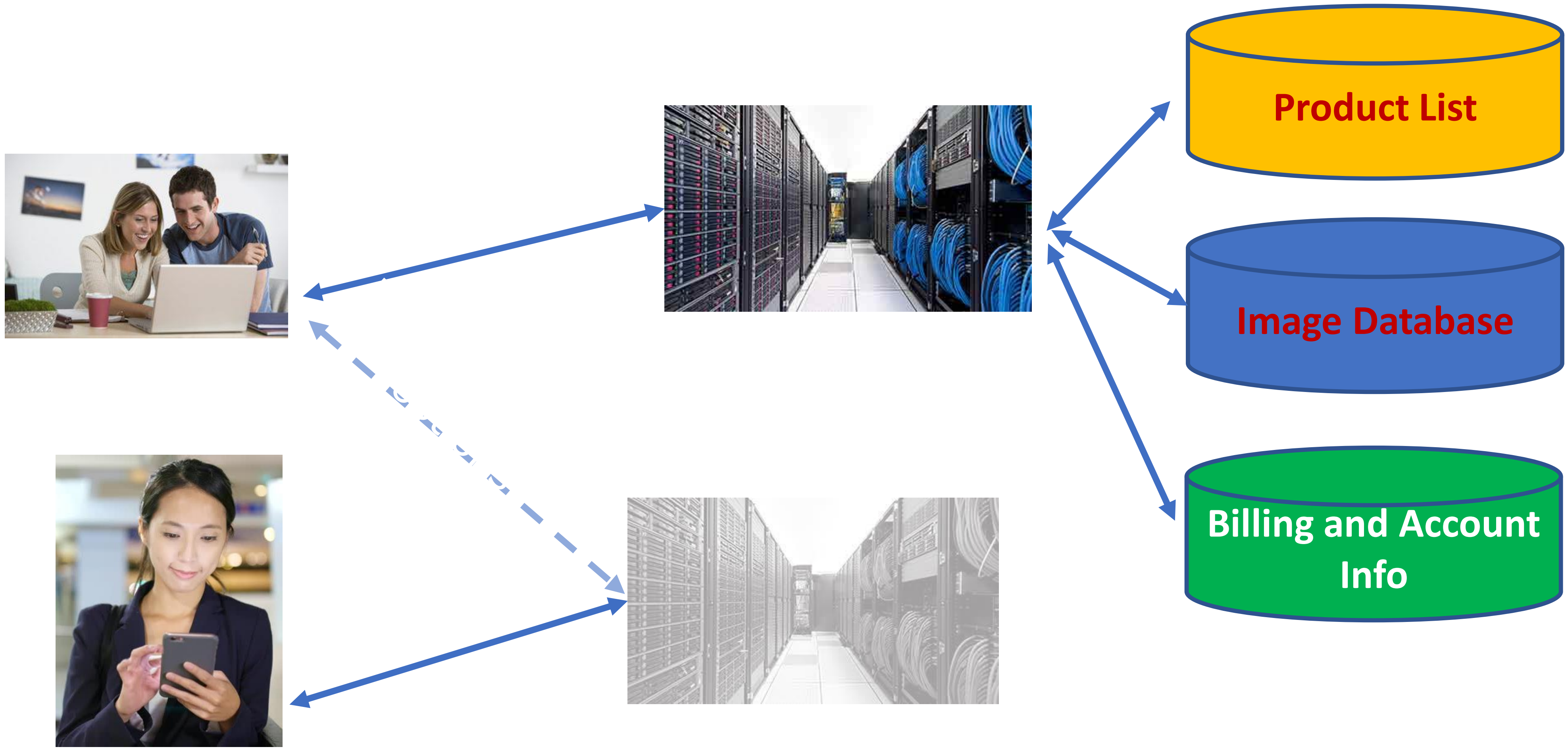
Old Approach (2005)



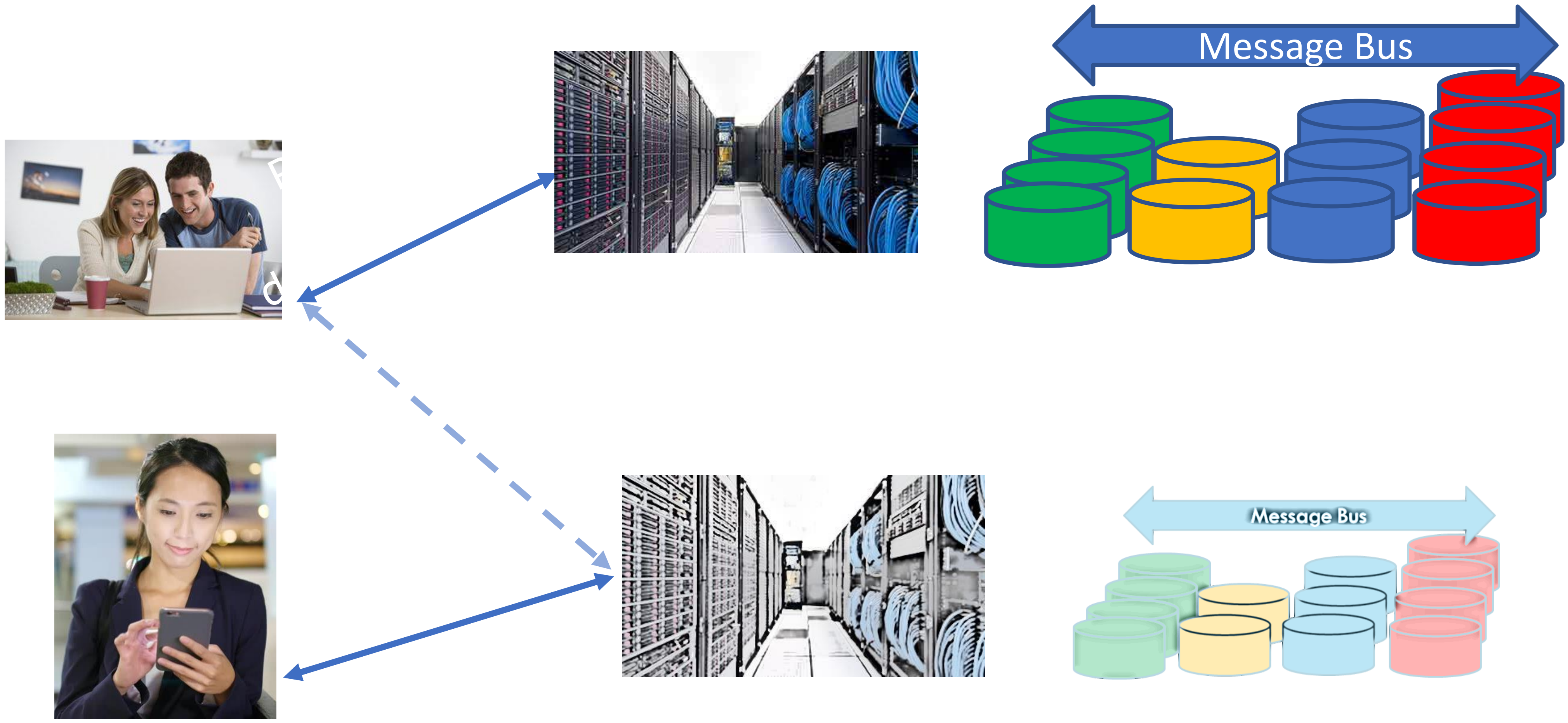
New Approach (2008)



New Approach (2008)



New Approach (2008)



Tier one / Tier Two

- We often talk about the cloud as a “multi-tier” environment

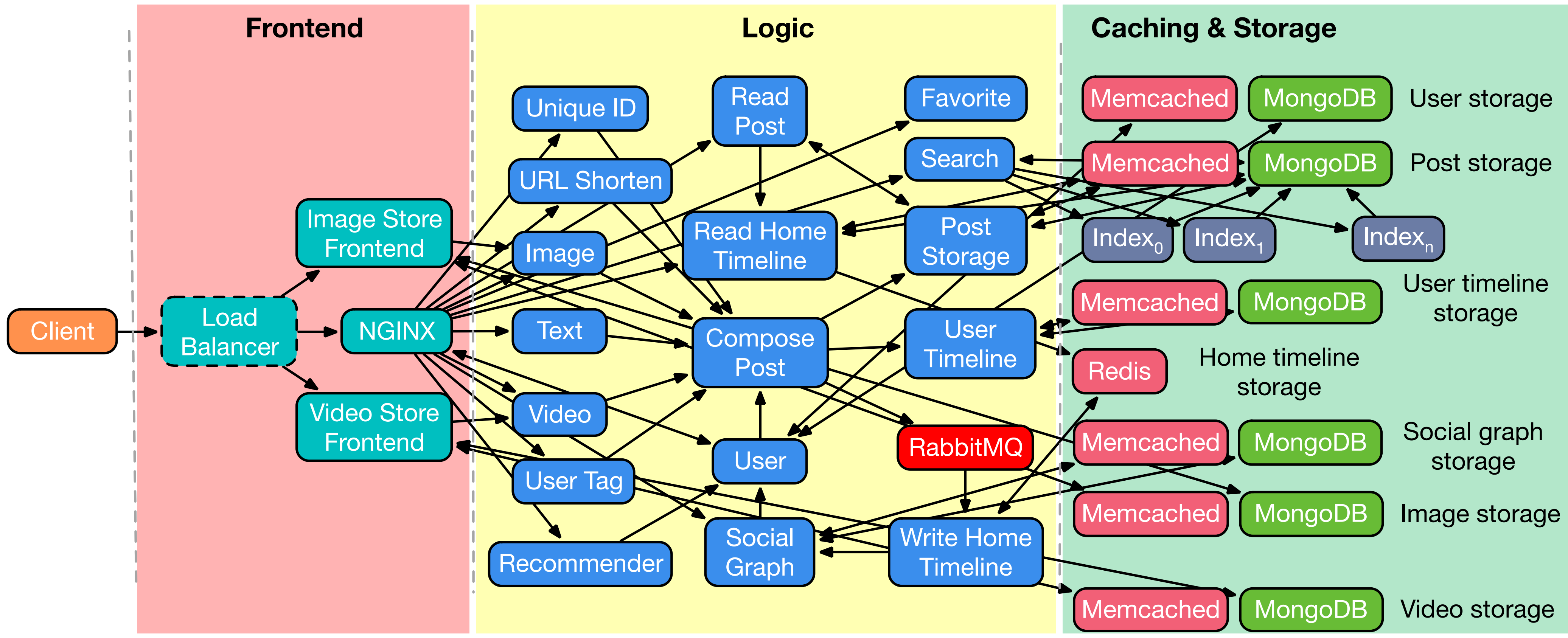
- Tier one: programs that generate the web page you see
- Tier two: services that support tier one

Today’s Cloud

- Tier one runs on lightweight servers:
  - They use small amounts of computer memory
  - They don’t need a lot of compute power either
  - They have limited needs for storage, or network I/O

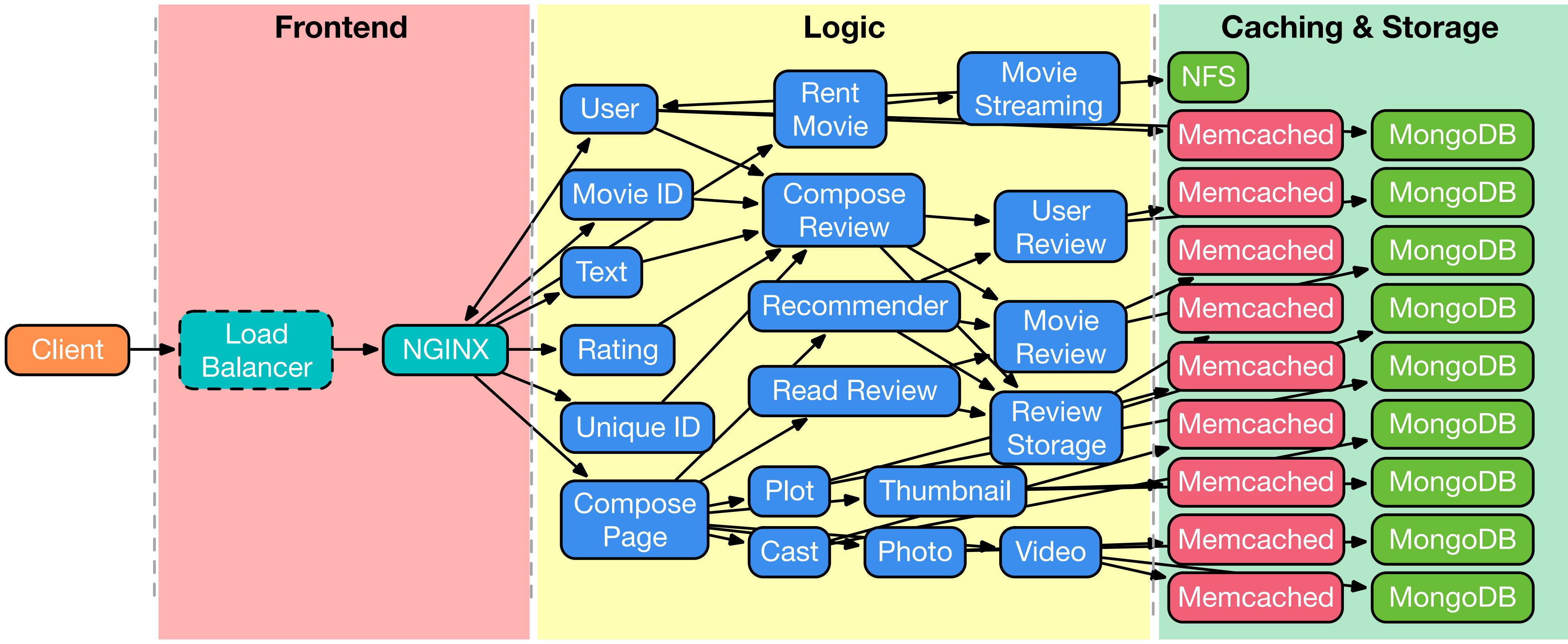
- Tier two run on somewhat “beefier” computers:
  - Provide many different microservices
  - Each microservice specializes in various aspects of the content delivered to the end-user

Microservices for Social Network



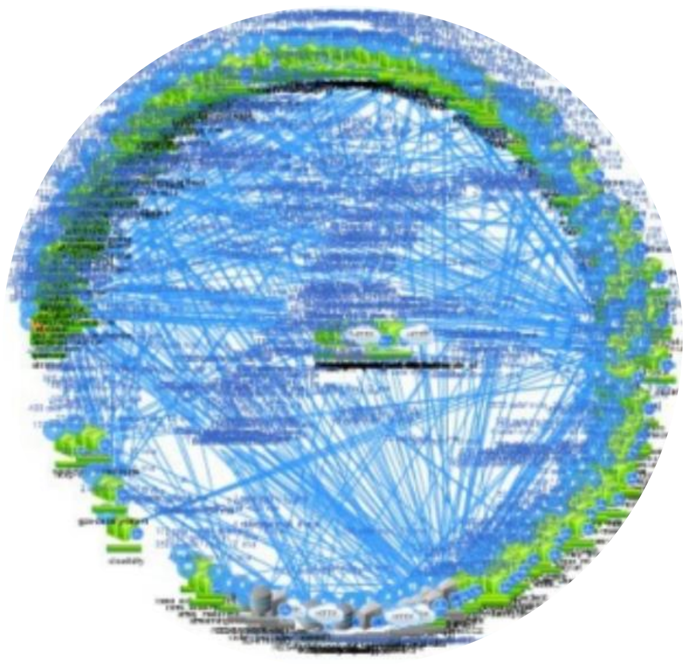
from Christina Delimitrou

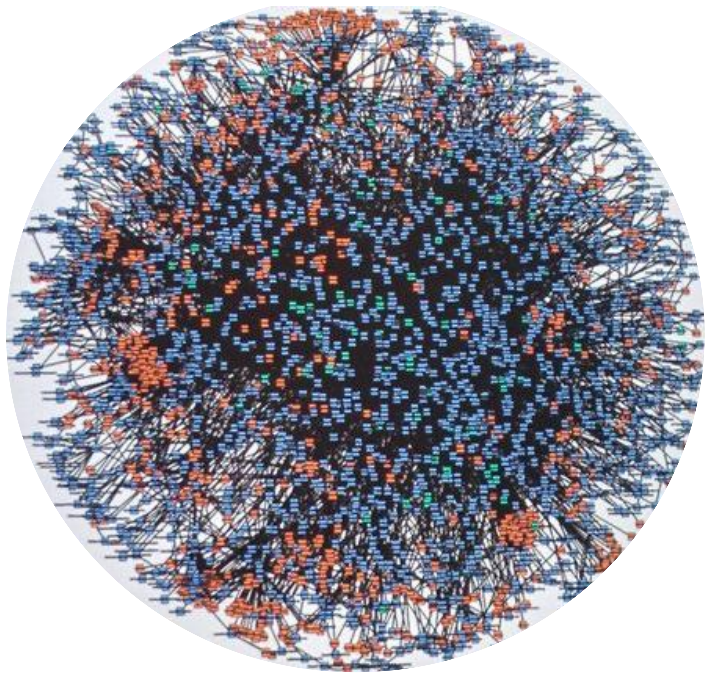
Microservices for Media Service



from Christina Delimitrou

Microservices Visualized





Each Microservice is a Parallel Pool!

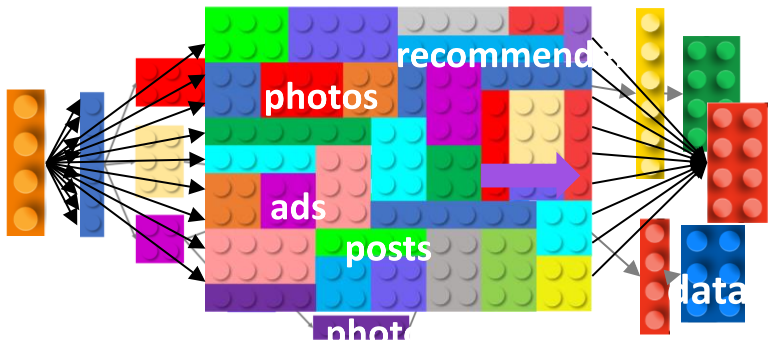
- Every one of those little nodes is itself a small elastic pool of processes
- A microservice is a program designed so that the data center can run one or many instances “elastically” to deal with dynamically varying demand
- The idea is that any instance can handle any request equally well, so there is no need for very careful “routing” of specific requests to specific instances
- This lets the data center adapt to changing loads easily!
- Load can vary significantly over time, so elasticity is critical, perhaps key defining feature of modern cloud

Pools are Managed Automatically

- Azure has App Service tool
  - Manages a large collection of compute resources in the cloud
  - Developers can install their own services as containers
  - Configuration files tell App Service when to launch service automatically
  - App Service can watch the queue of requests and automatically add instances or shut instances down to match loads
- Kubernetes
  - Provides similar functionality as App Service but more complex because it provides much finer-grained control
  - Especially useful for complex, microservice style apps

Benefits of Microservices

- Advantages of microservices
  - Modular, so easier to understand
  - Helps speed development & deployment
  - On-demand provisioning, elasticity
  - Language/framework heterogeneity



from Christina Delimitrou

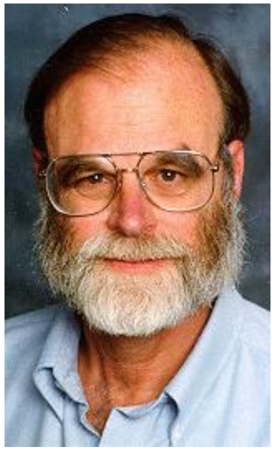
Questions about Data

- All the microservices access data, cache data, update data, replicate data
- Can we ensure that data is accessed correctly and consistently, even in the presence of failures?
- Solution: use a single BIG database, replicate it for fault tolerance

Single Big Service Performs Poorly

- Until 2005 “one server” was able to scale and keep up, e.g., 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 didn’t work
- 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 scaling issue as early as 1996
- He and colleagues wrote a wonderful paper based on 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.

- Basic message: divide and conquer is really the only option

Approach in the Paper

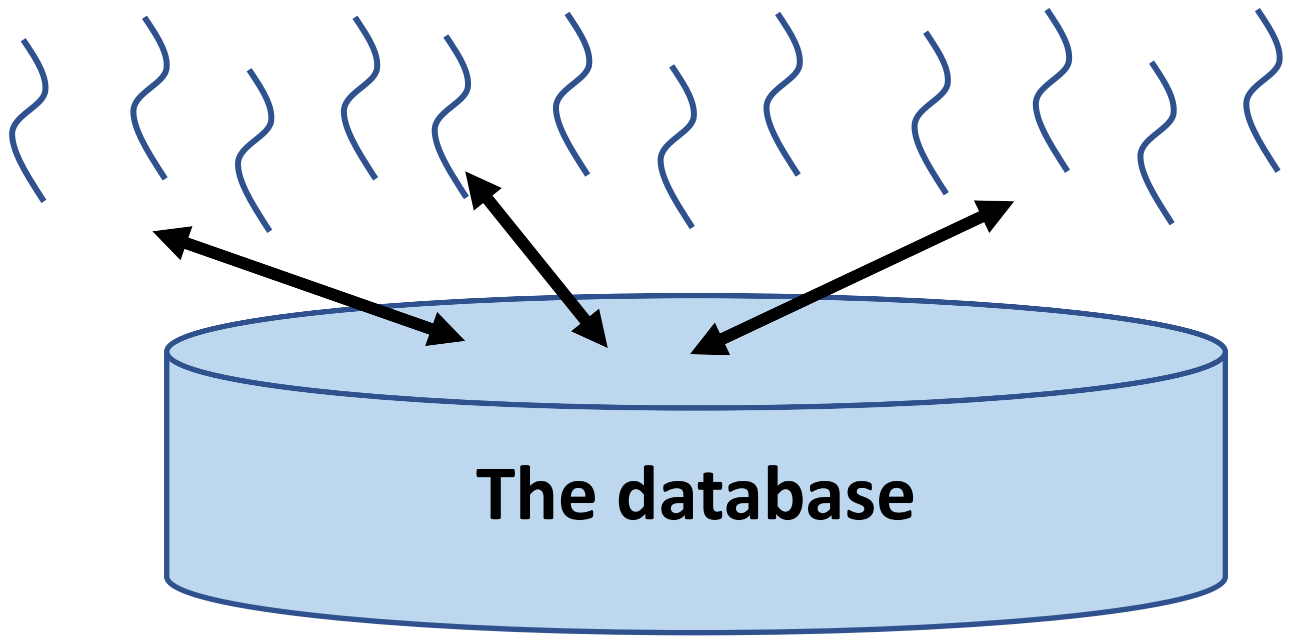
- The paper uses a “chalkboard analysis” to analyze scaling for a replicated system that behaves like a database
  - Analysis based on asymptotic costs, similar to complexity analysis
- System could be an actual database like SQL Server or Oracle
- But their “model” also covered any other storage layer that provides strong guarantees of data consistency

System Model

- The paper assumes that the service provides some form of lock-based consistency, which they model as database serializability
  - Applications use read locks, and write locks
- System uses a pool of replicated servers
  - Work is spread across servers
  - Enables handling increasing application load

Their Setup

Applications using the database are client processes

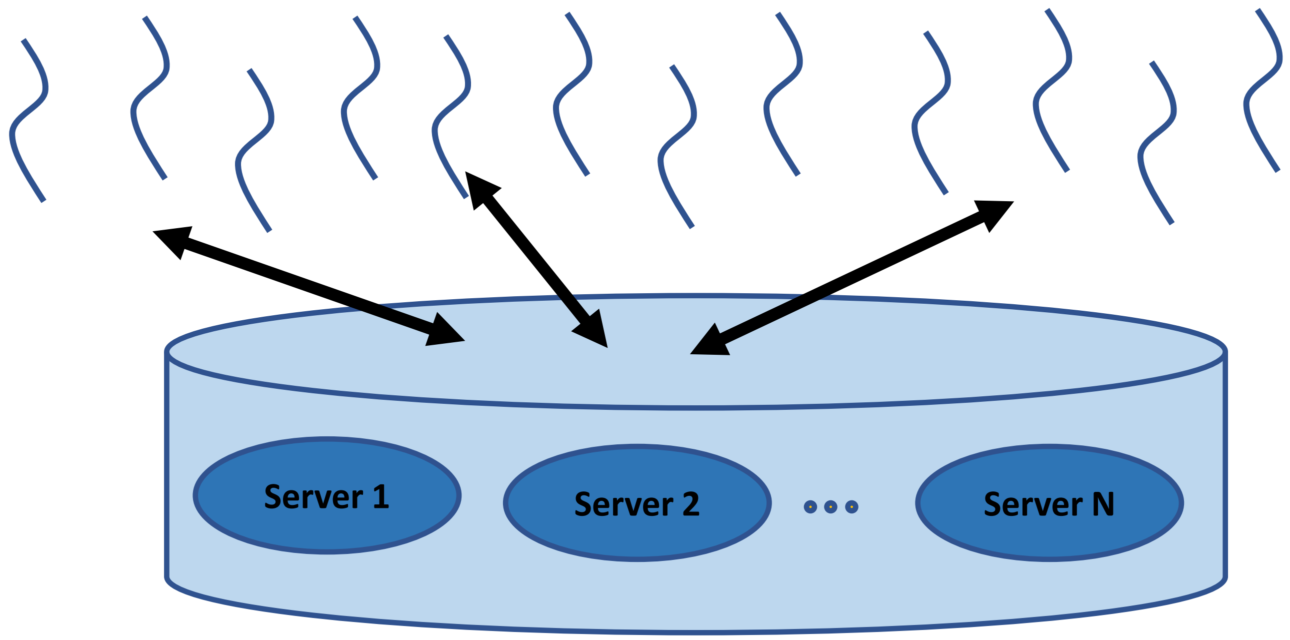


During the run, T concurrent transaction requests are issued.

Here, 3 are running right now, but T could be much larger.

Their Setup

Applications using the database are client processes



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

Their Analysis

- Goal: A scalable system with N servers should be able to handle N times more transactions (T)
- Instead, they found that the work the servers must do increases non-linearly with N
- One reason is that each update must be replicated to all N servers
  - So, node update rate (across all nodes) grows as N^2
- Worse, deadlocks occur as N^3, causing feedback (because reissued transactions get done multiple times)
  - 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

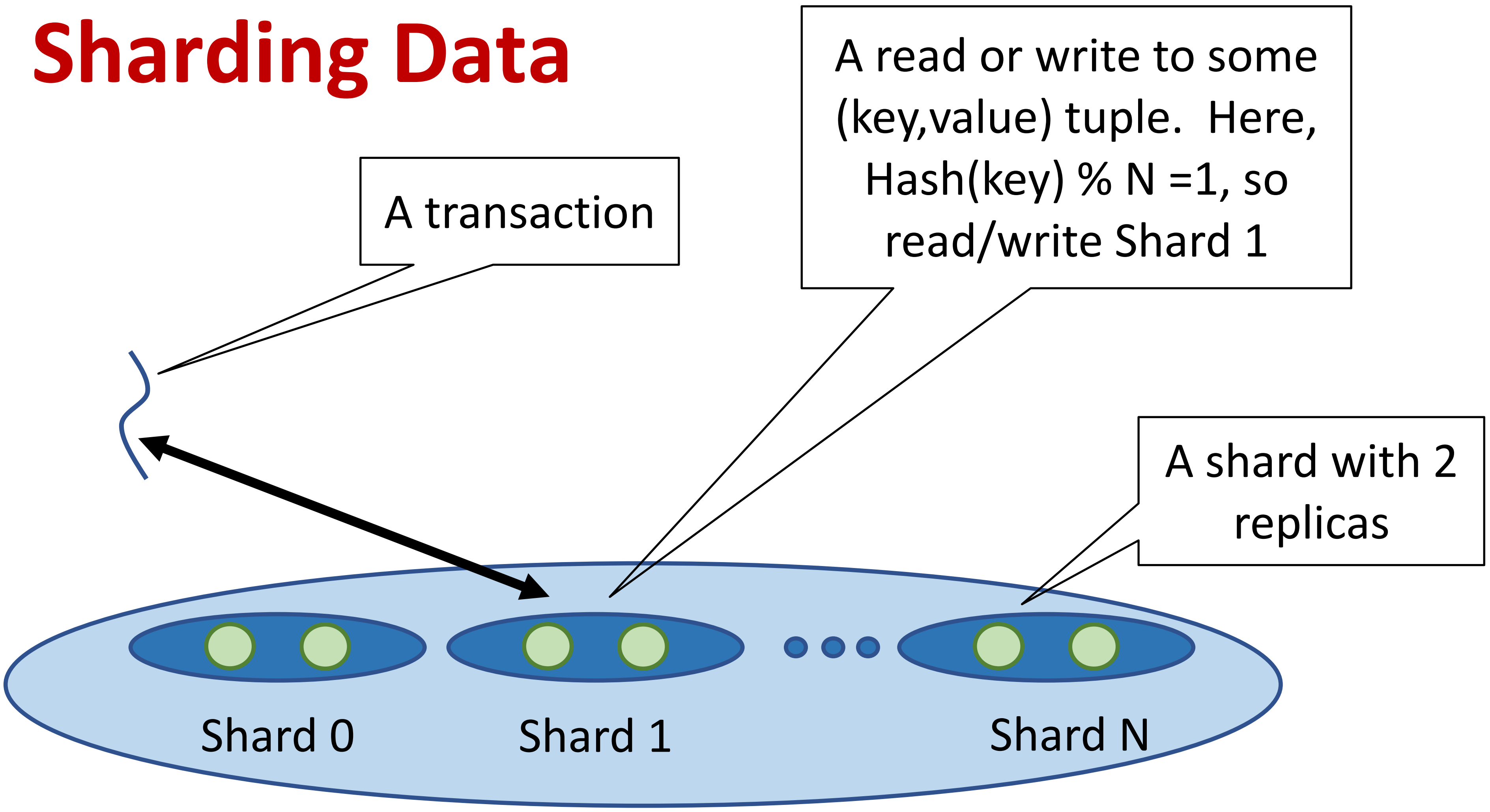
Why do Services Slow Down at Scale?

- The paper pointed to several main issues:
  - Lock contention: with more concurrent transactions, they are more likely to try to access the same object and wait for locks
  - Abort: deadlock also causes abort/retry sequences, some consistency mechanisms use optimistic behavior, but now and then, they must back out and retry
- The paper explores many options for replication schemes but ends up with similar negative conclusions
- These conclusions may seem database-specific, but these issues arise in any service that provides consistent data

So, How Should Services be Scaled?

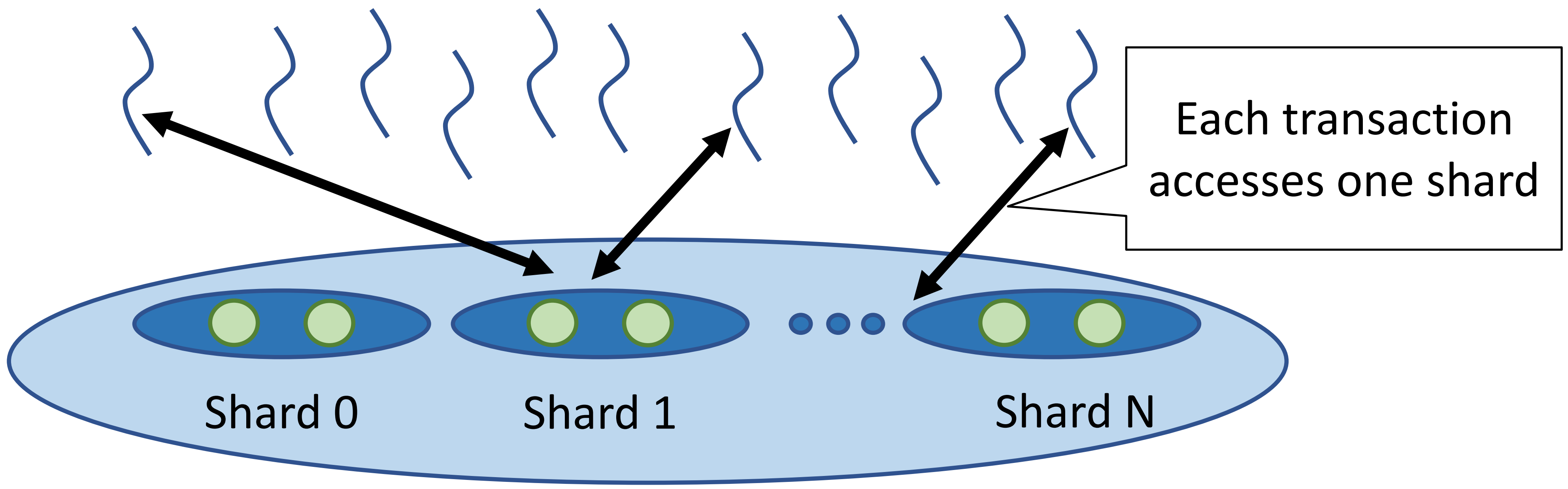
- Back in 1996, Jim’s paper concluded that you need to shard the database into a large set of much smaller databases, with each storing distinct data
- Jim set out to do this for a massive database of astronomy data
- By the time he died in 2007, Jim had shown that for every problem he ran into, it was possible to devise a sharded solution in which transactions mostly touched a single shard at a time
- In 1996, it wasn’t clear that every important service could be sharded, by the 2007 period, Jim had made the case that in fact, this is feasible!

Sharding Data


A transaction
A read or write to some (key,value) tuple. Here, Hash(key) % N =1, so read/write Shard 1
A shard with 2 replicas
Shard 0
Shard 1
Shard N

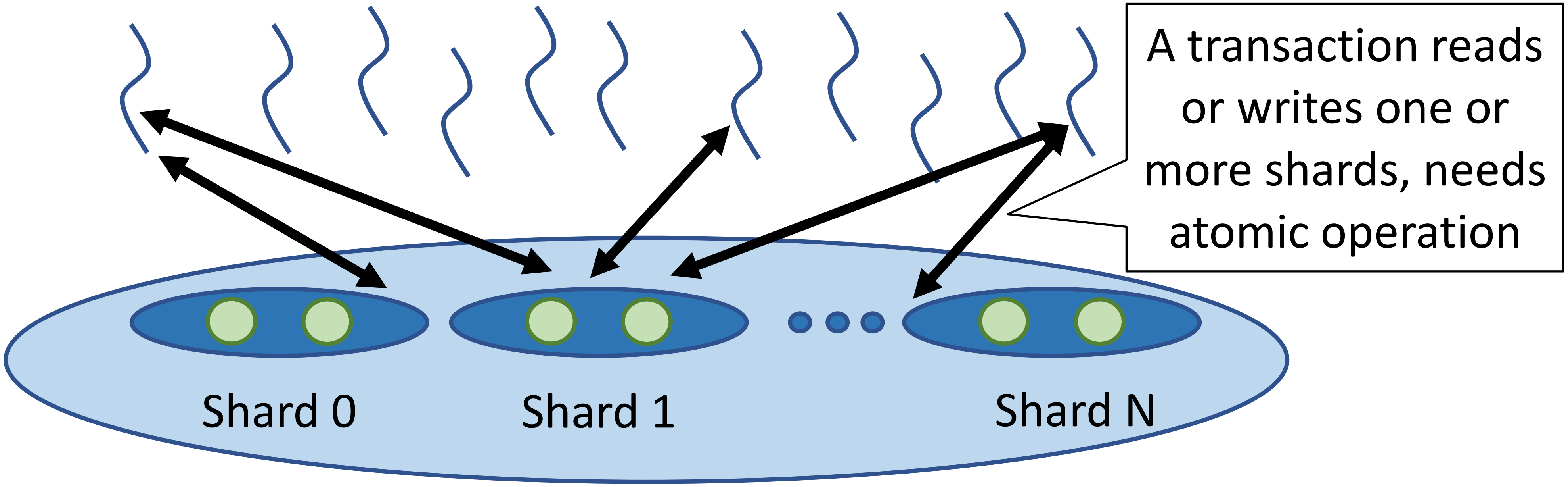
Sharded storage service with N shards,
2 replicated servers per shard

Sharding with Single Node Transactions



If each transaction does all its work at just one shard, never needing to access more than one shard, then sharding scales well

Sharding with General Transactions



A transaction reads or writes one or more shards, needs atomic operation

Transactions that touch multiple shards hold locks for long time, need 2-phase commit (agreement protocol) for atomicity

In this case, Jim Gray's analysis applies, as we scale up, performance suffers

Example: A Microservice for Caching

- Let’s look at the concept of caching as it arises in the cloud, and at how we can make such a service elastic
- This is just one example, but is a good example because key-value structures are very common in the cloud
  - E.g., facebook uses elastic caching to cache binary large objects (i.e., pictures)

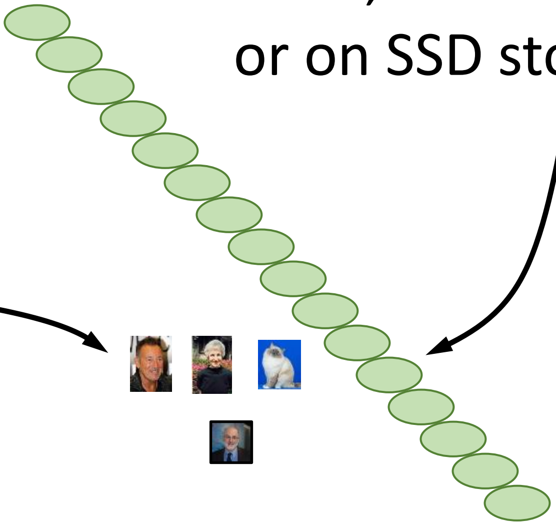
Sharded Caching

Each machine stores a set of (key,value) tuples in a local hash table, stored in DRAM or on SSD storage

Store(Key, Value)

Key=Birman	Value=

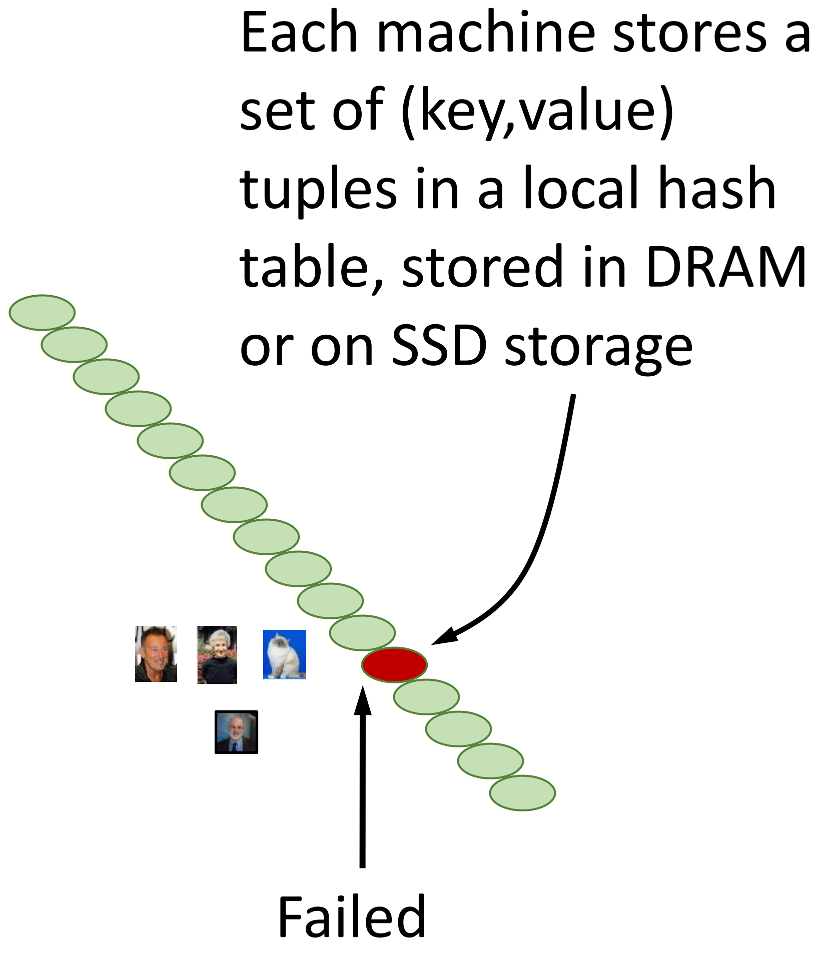
Hash("Birman") % N



In effect, two levels of hashing!

Failures with Sharded Caching

- What if process or machine storing cached data fails?
- A portion of the cache would not be available
- Data can still be fetched from backend database server, but this adds load on the backend, increases tail latency

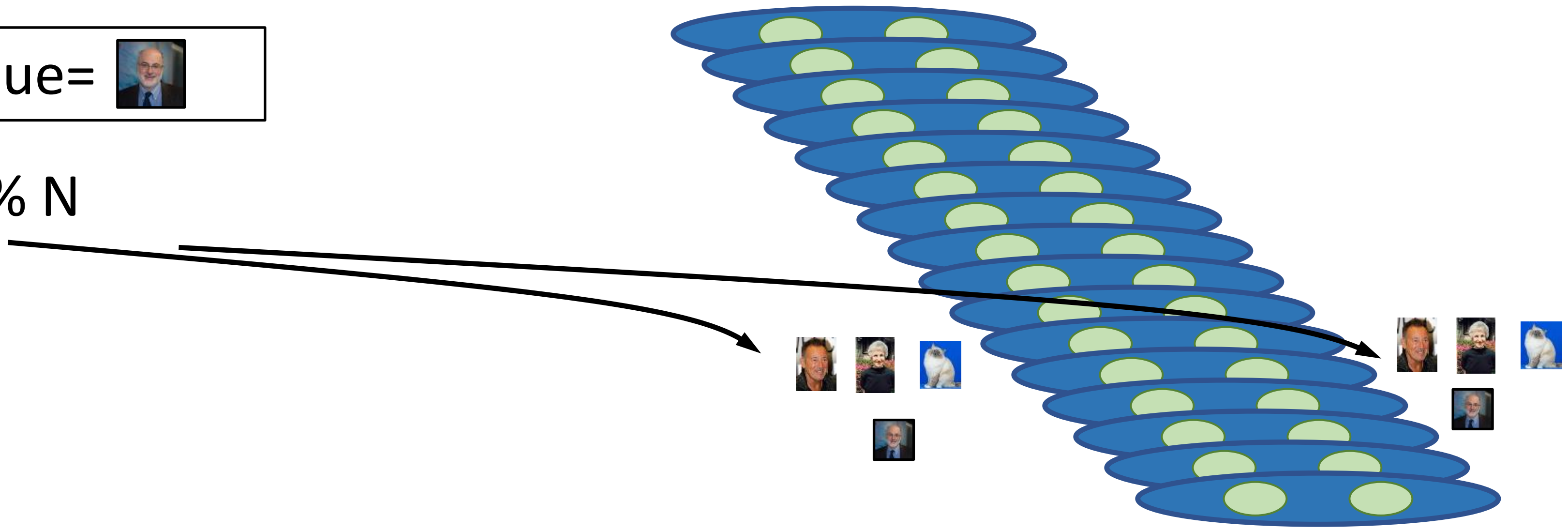

Each machine stores a set of (key,value) tuples in a local hash table, stored in DRAM or on SSD storage
Failed

Replicated, Sharded Caching

Store(Key, Value)

Key=Birman	Value=

Hash(“Birman”) % N



N shards, each shard shored on two machines,
i.e., two machines store the same set of (key,value) tuples

Terminology

- This design is called a key value store (KVS) or a distributed hash table (DHT)
- A distributed KVS contains shards or partitions of data, and each shard may be replicated

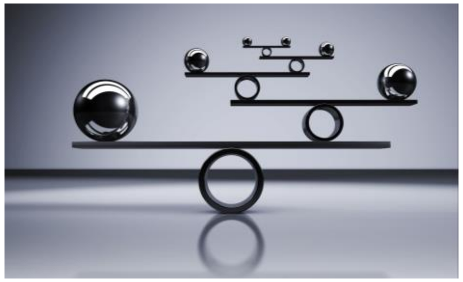
Typical KV Store API

- The MemCached API was the first widely popular KV Store
- Today there are many important KV store, e.g., MemCached, RocksDB, TigerDB, DynamoDB, BigTable, Cassandra, and the list just goes on and on
- Most support some form of

```
put(key, value)  // store (key, value)
value = get(key) // get the value associated with key
watch(key)       // notify when the value is updated
```

- Some hide these basic operations behind file system APIs, or publish-subscribe APIs, or database APIs

Load Balance in KVS



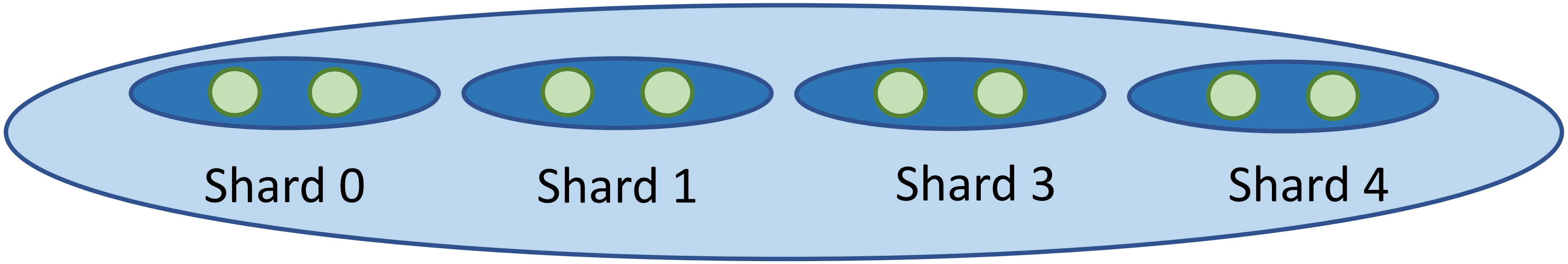
- Hashing stores keys on different servers
- So, do all servers stores similar number of keys?
- Depends on the hash function?
  - Secure hash functions such as SHA-256 are relatively fast and generate random output, so keys are spread uniformly
  - Other hash functions may not spread data uniformly
- Even if keys are stored uniformly across server, can there be load imbalance?
  - Yes, some keys may be heavily accessed, causing hot spots
  - One solution is to have N (replicated servers) storing KN shards
  - It is unlikely that all K shards on a server will be loaded

Elasticity Adds Another Dimension

- If we expect changing load patterns, the cache may need a way to dynamically change sharding policy
- Since a cache "works" even when empty, we could simply shut it down and restart with a different number of servers and another sharding policy.
- But cold caches perform poorly
- Instead, we should ideally "shuffle" or reconfigure cached data across servers

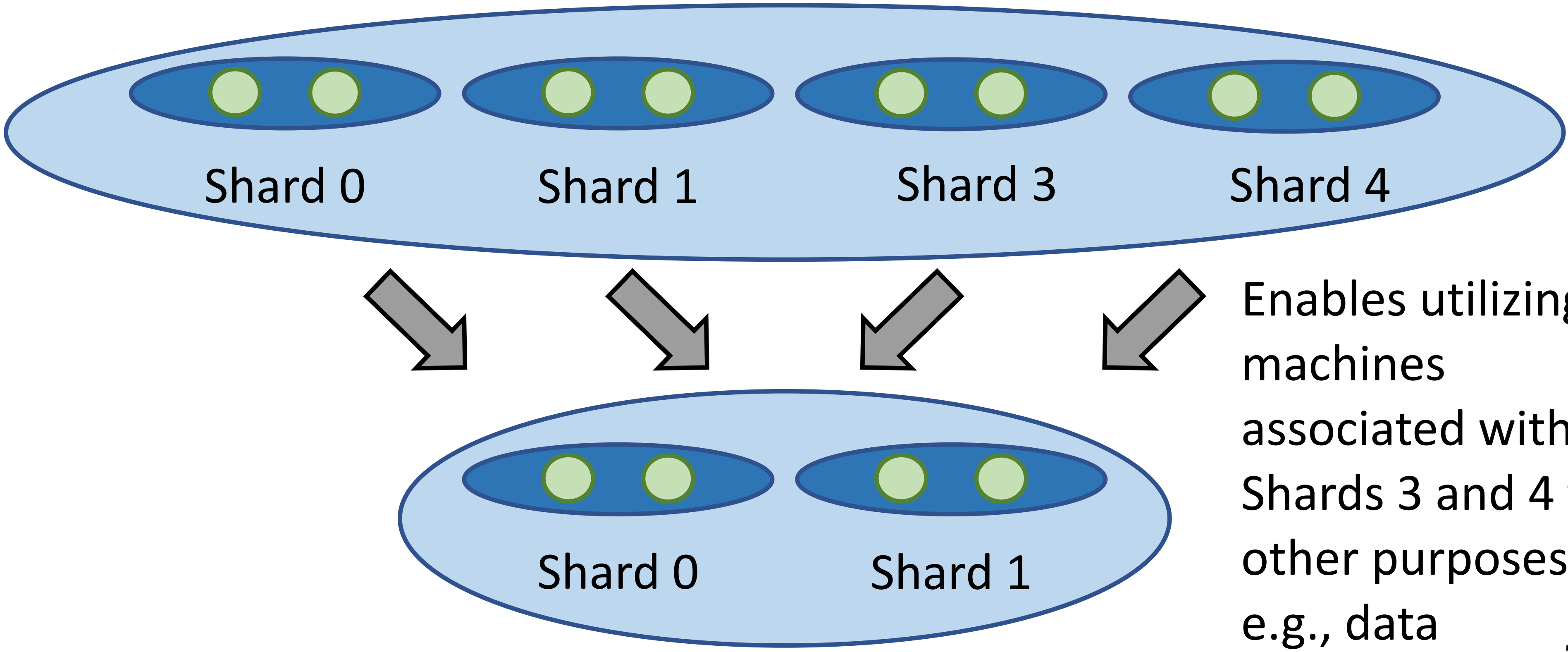
Elastic Shuffle

- Say, we initially had the cache data spread over 4 shards



Low Load

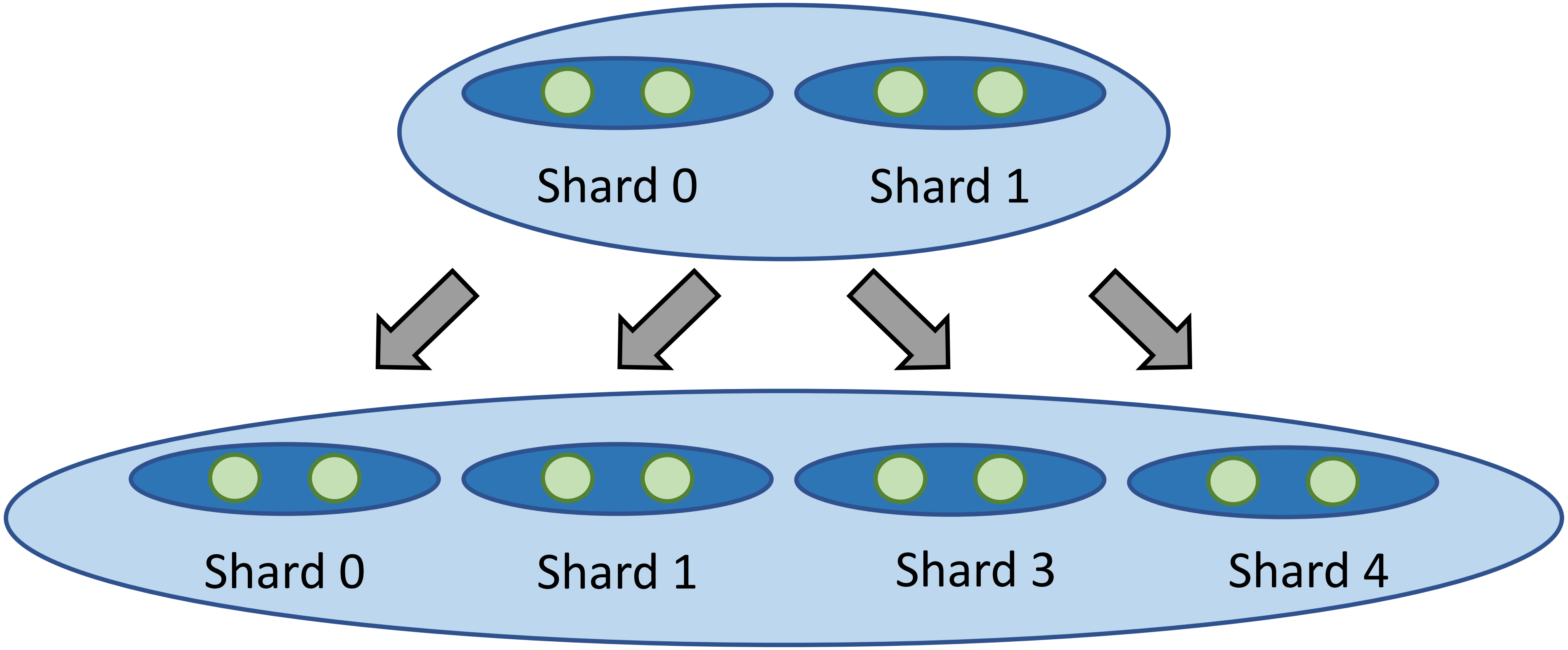
- During low load, we could move the cached data into 2 shards, while dropping half the cached items
    - Hopefully, keep the more popular items



Enables utilizing machines associated with Shards 3 and 4 for other purposes, e.g., data analytics

High Load

- During high load, we could add machines and shuffle data to expand the cache



Where is the Cache?

- How can applications or other services that use the cache find out the location of the caching machines?
- Typically, big data centers have a management infrastructure service that keeps this type of configuration information
  - E.g., list of processes@machines that cache the data, parameters needed to compute the mapping from the key to the shard, shard replicas, etc.
- When this configuration information changes, applications are told to re-read the configuration
- Later, we will learn about one such service, Zookeeper

Data Consistency Issues

- Many reasons for inconsistent data accesses
  - Caches are inconsistent with storage, e.g., some clients bypass cache and access storage directly
  - Storage replicas are inconsistent
  - Caches are replicated and inconsistent
- Strong consistency ensures that reads return the latest write, making it easier to write applications
  - While data sharding helps scaling the database, strong consistency can limit availability and scalability

What About Weak Consistency?

- Some tasks may precompute data (“last night”) and use this read-only data
  - Read potentially stale data, better than read no data
- They can also enqueue update tasks for offline processing
  - Allow delayed updates, better than disallow updates
- Tasks might also guess the effect of updates, but the offline version will “win” if a conflict occurs
  - Buy an item, eventually told it was sold out, get refund

In the Cloud, Not Every Subsystem Needs the Strongest Guarantees

- At Berkeley, Eric Brewer argued that strong consistency delays response
  - For example, conflicting database updates can be forced into an agreed order, but this takes time and involves node-node dialog, and if there is a network partition, the system provides no availability
- But services make money only when they always provide fast response
- Eric concluded that this means cloud services may need to relax consistency
- This insight is captured in his CAP rule (Consistency, Availability and Partition Tolerance)

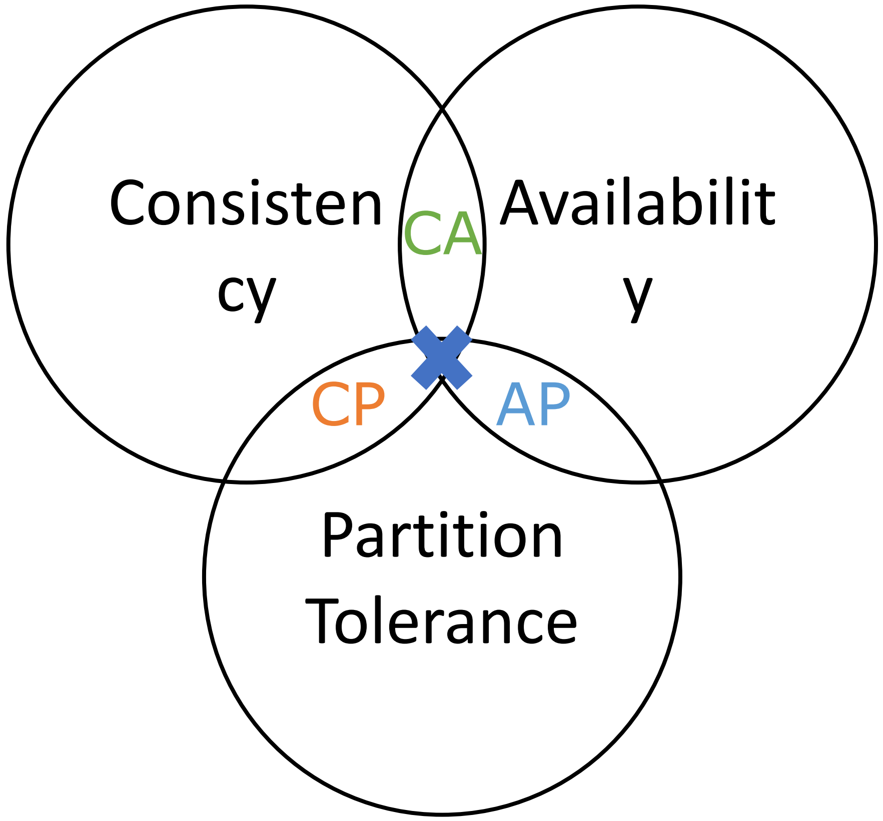
Definitions, Slightly Informal!

- Consistency: Updates are performed in some system-selected order by all replicas. Queries return most up-to-date values. Users see a single system.

- Availability: The system responds to every user request, even when some machines are down.

- Partition Tolerant: The system can tolerate network failures between subsystems. E.g., machines are partitioned into separate subnets and the switch between the subnet fails.

Cap Rule



- You cannot achieve all three of:
  - Consistency
  - Availability
  - Partition-Tolerance

- Popular interpretation: choose 2-out-of-3
  - CA: Assumes partitions don't occur, not realistic
  - CP: poor availability, users unhappy
  - AP: hard to program, possibly confusing to users

- None of these options are appealing!

CAP rule in practice

- Partitions do occur, so systems must be partition tolerant
  - You cannot not choose partition tolerance …
  - But you can design systems to make them rare
- When there are no partitions,
  provide both consistency and availability
- When there is a partition, systems need to choose between consistency vs. availability, e.g., design systems that are best suited for application’s consistency and availability needs
- When partition is fixed, restore consistency and availability, e.g., reconcile inconsistent replicas

BASE Methodology

- BASE: A set of rules for implementing CAP-based solutions
- Invented at eBay, adopted by Amazon, others
  - Basic Availability: provide continuous availability, despite failures or temporary inconsistency
  - Soft State: use state that can be regenerated (e.g., cached data) for efficiency
  - Eventual Consistency: assuming no further updates to an item, all users will eventually see the same value of the item
- Soft state and eventual consistency help recovery from failures, network partitions, data inconsistency, etc.

BASE Example

- For example, if product photos rarely change, cache them, do not check for staleness with each cache access, let them expire after a few weeks
  - Avoids all cache refresh traffic
  - If a photo does change, you do see a stale product photo, but this is rare, and it will be refreshed eventually

- BASE ≡ “CAP in practice”
  ≡ “Use CAP. You can clean up later.”

- BASE encourages developers to think about when they need or do not need consistency

Thanks!

- Please go over the class web page available from http://www.eecg.toronto.edu/~ashvin
- Please use Quercus Discussions for any class related questions