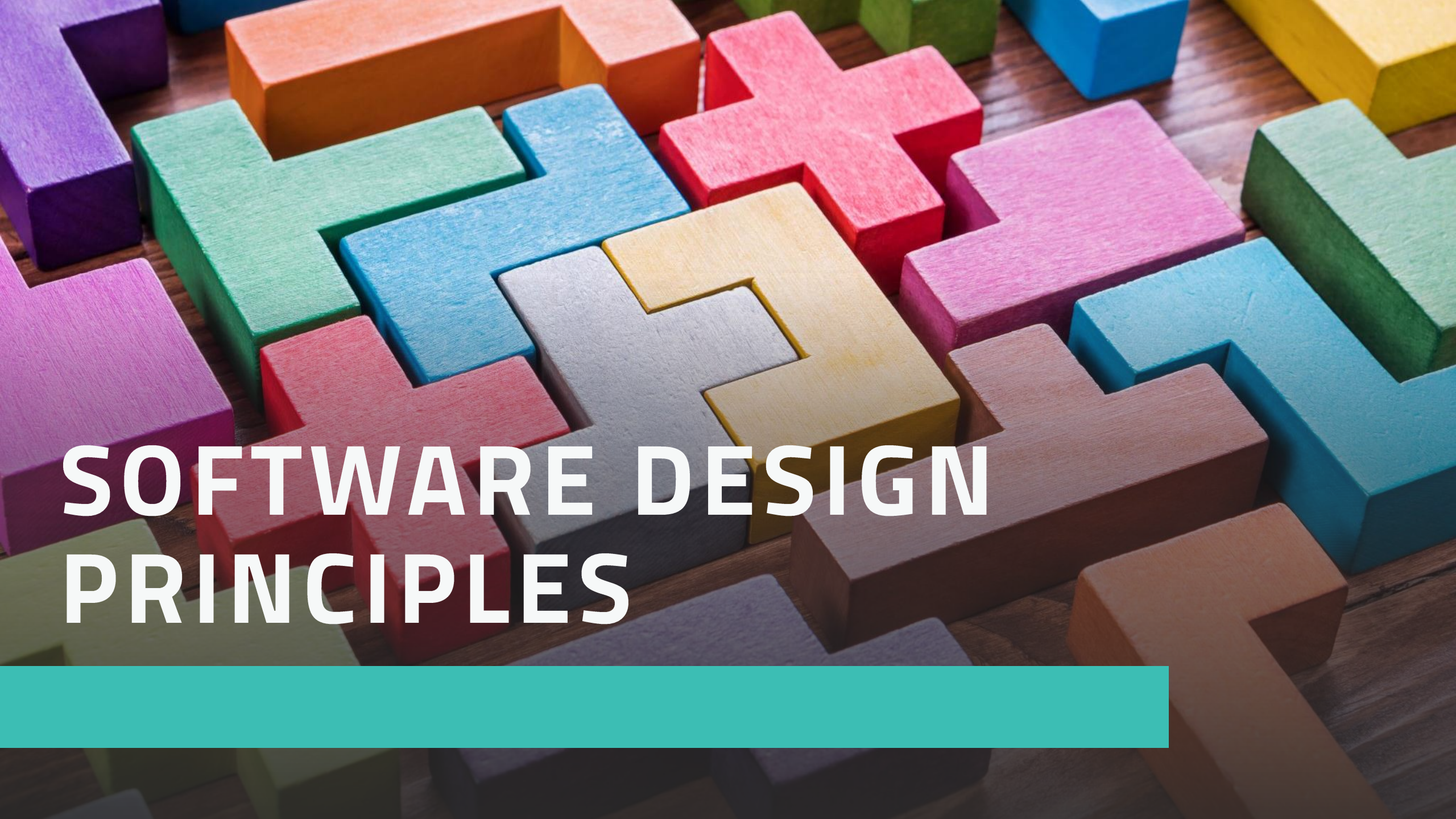
MASTERING THE SYSTEM DESIGN INTERVIEW

BY FRANK KANE

Sundog Education

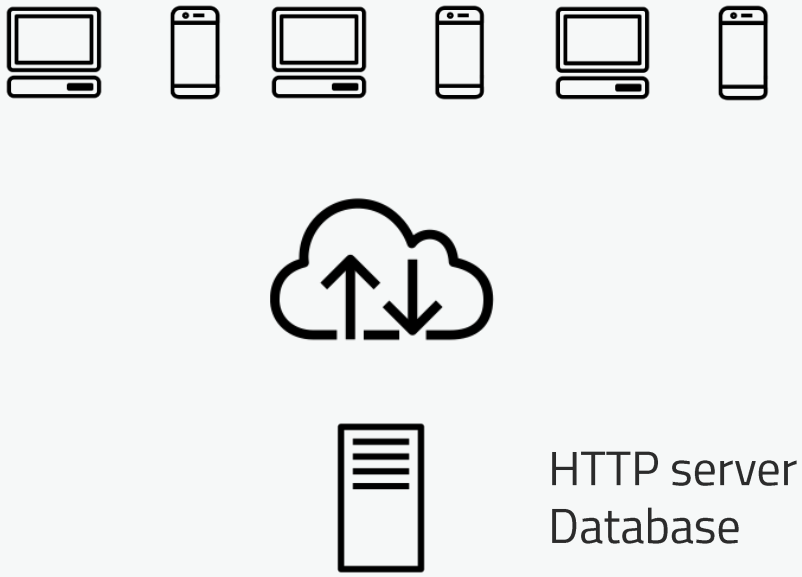
Why listen to me?



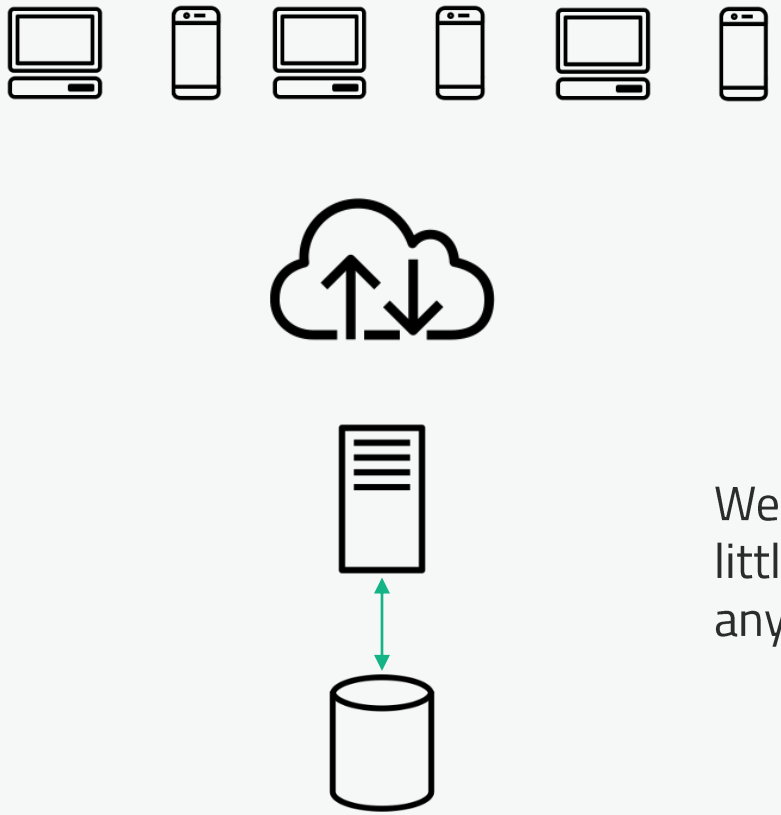
SOFTWARE DESIGN PRINCIPLES


SCALABILITY

Single-Server Design

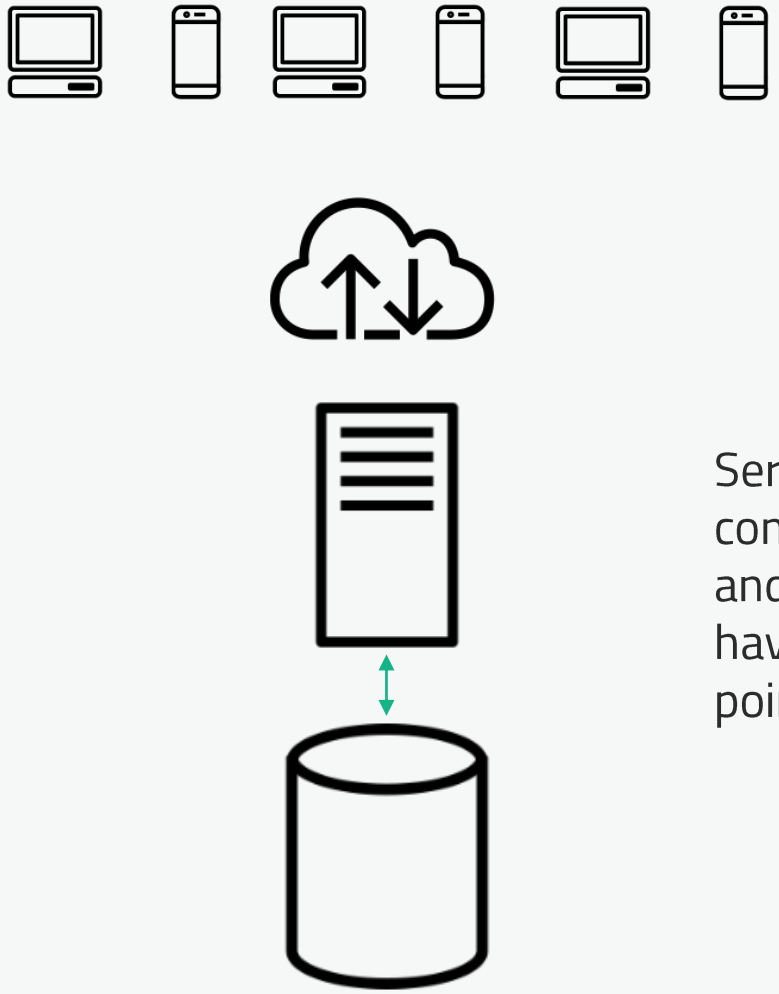


Separating out the database



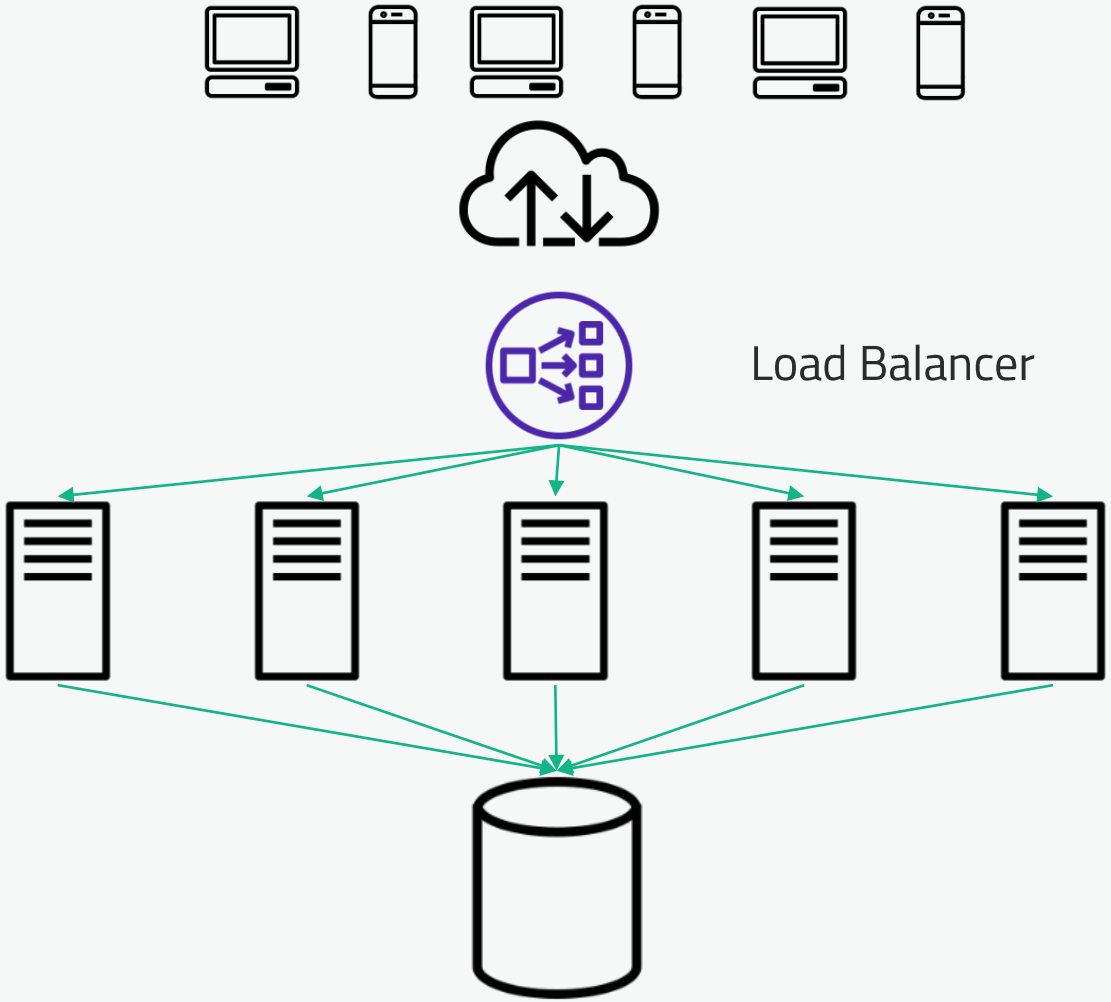
Well, that's a little better anyhow.

Vertical scaling



Servers only come so large, and you still have single points of failure.

Horizontal scaling



This is easier if your web servers are “stateless”.

Api calls are independent, i.e. if one API call requires to store some data on the server and 2nd API call tries to access it, Then we can't go for this architecture. Cause Load balancer may route the 2nd API call to another server

Choose the simplest architecture that meets your projected traffic requirements.

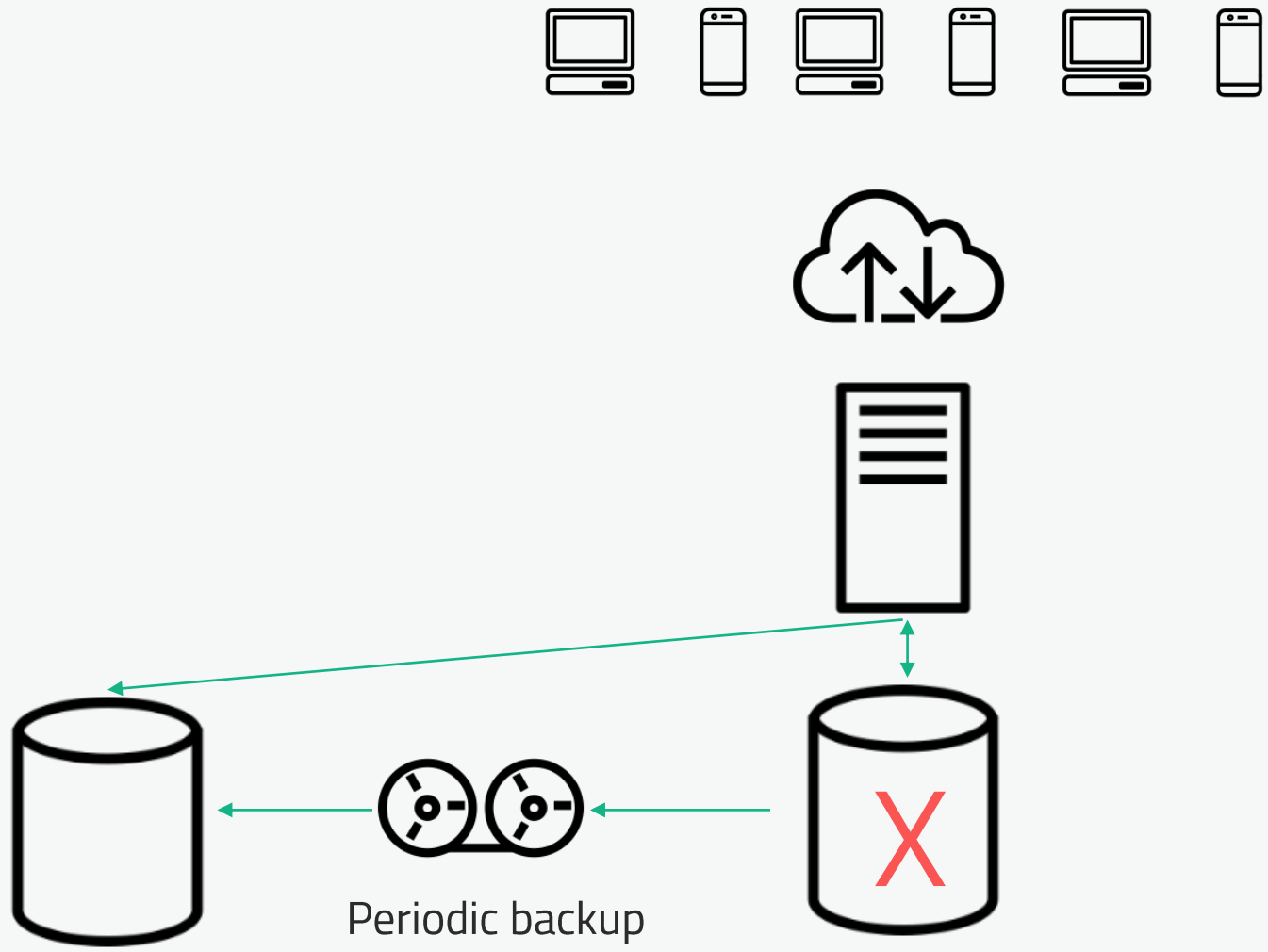
But no simpler.

Side note: Where do servers come from?

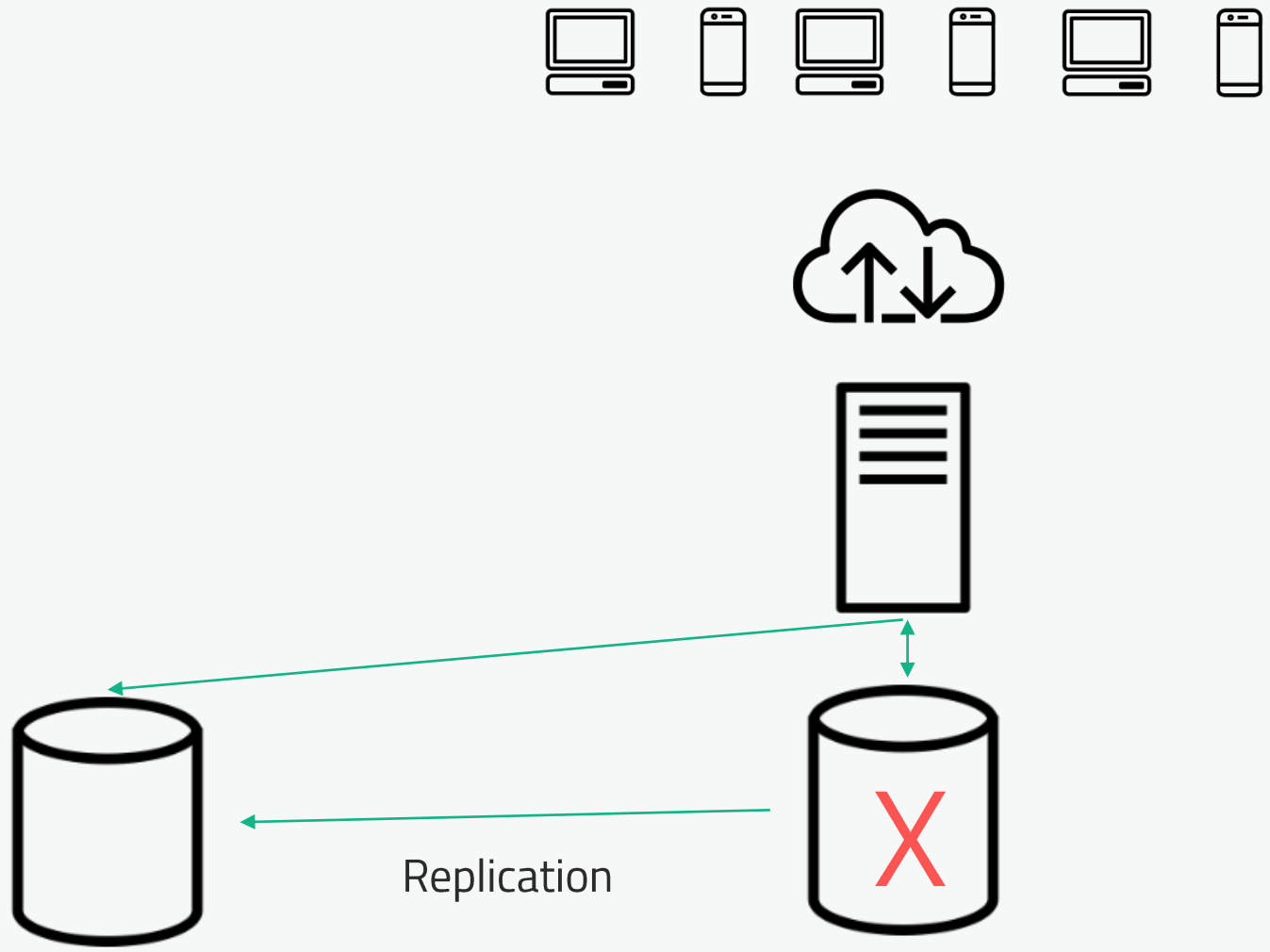
- Provisioned within your own company's data centers
- Cloud services (i.e., Amazon EC2, Google Compute Engine, Azure VM's)
- Fully managed "serverless" services (i.e., Lambda, Kinesis, Athena)

SCALING THE DATABASE

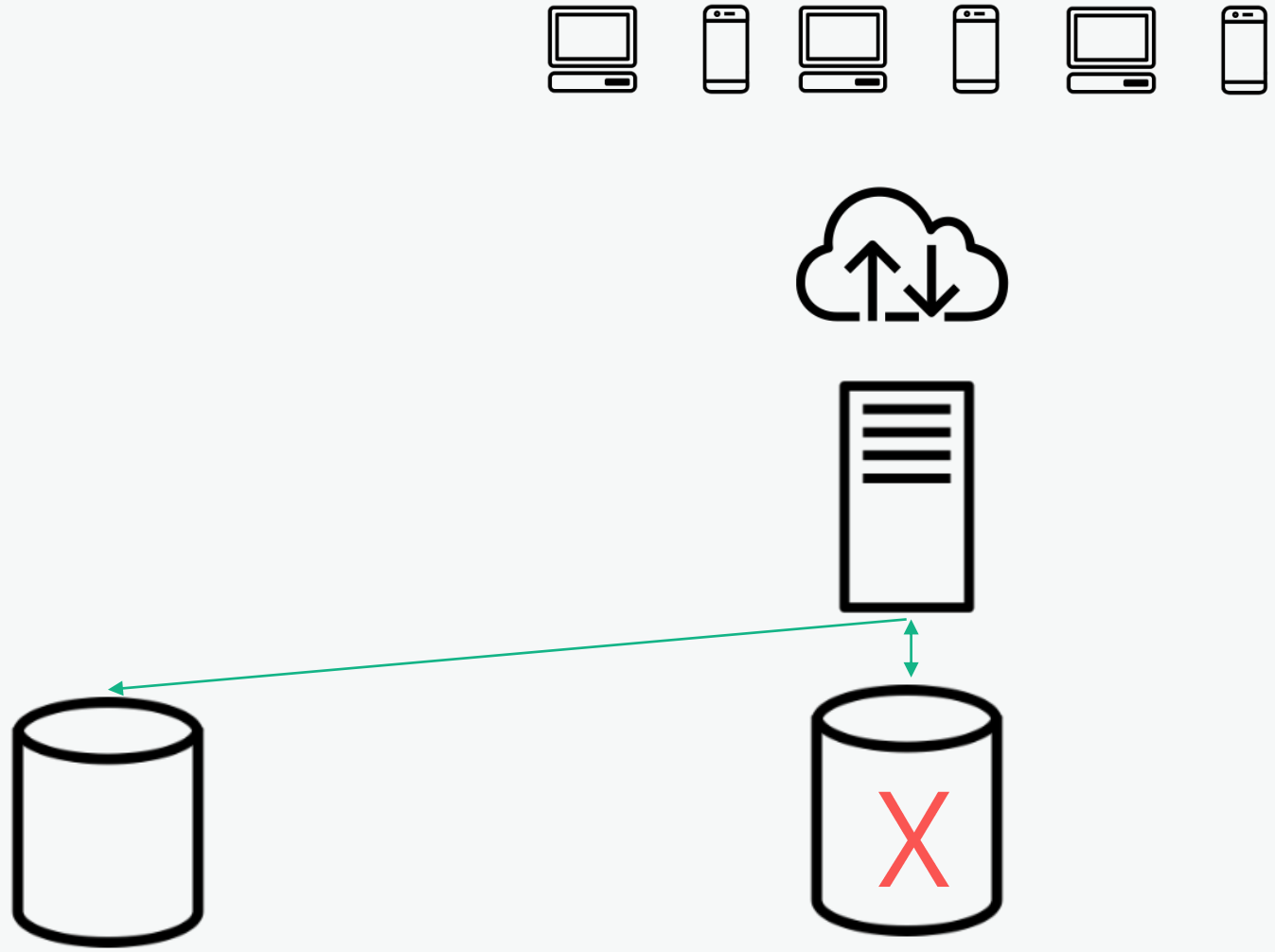
Failover servers: Cold Standby



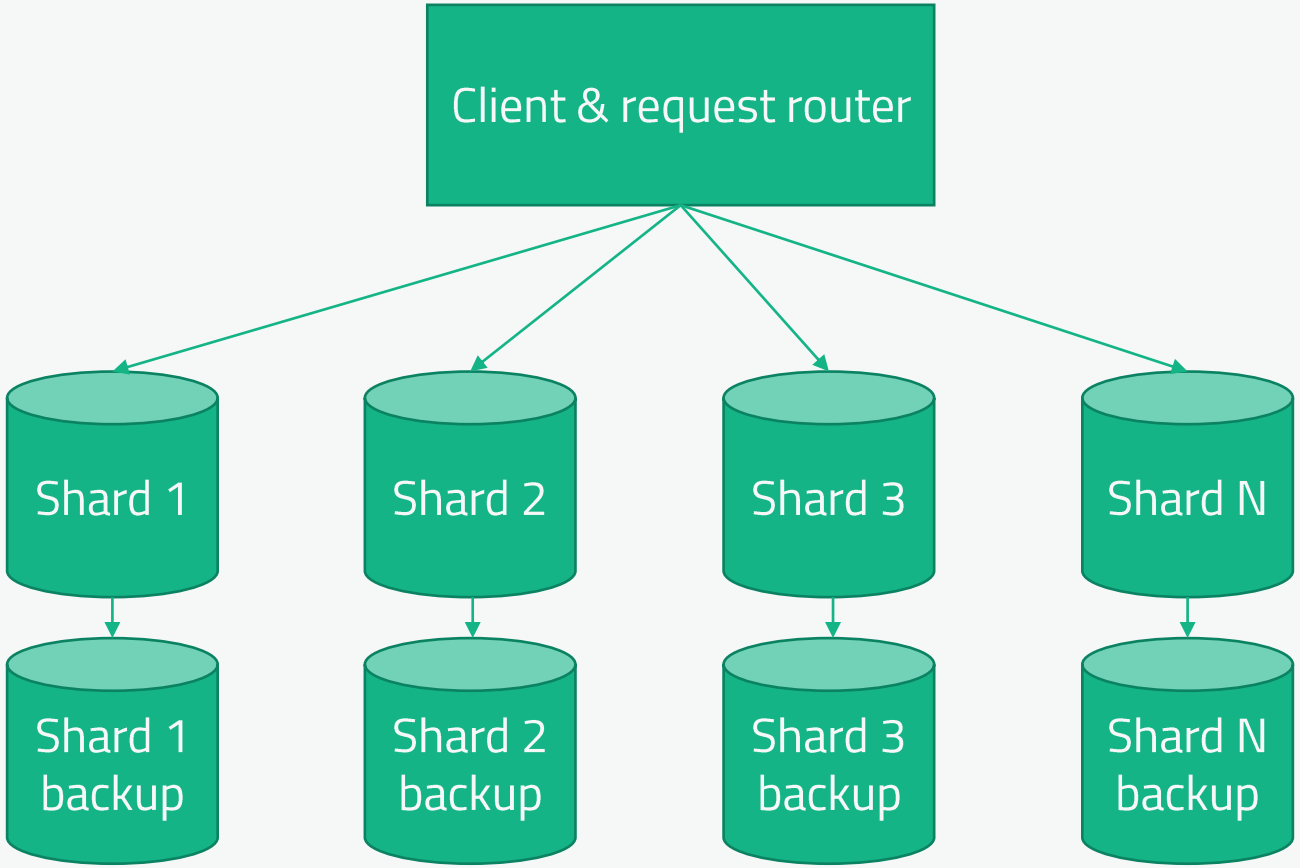
Failover servers: Warm Standby



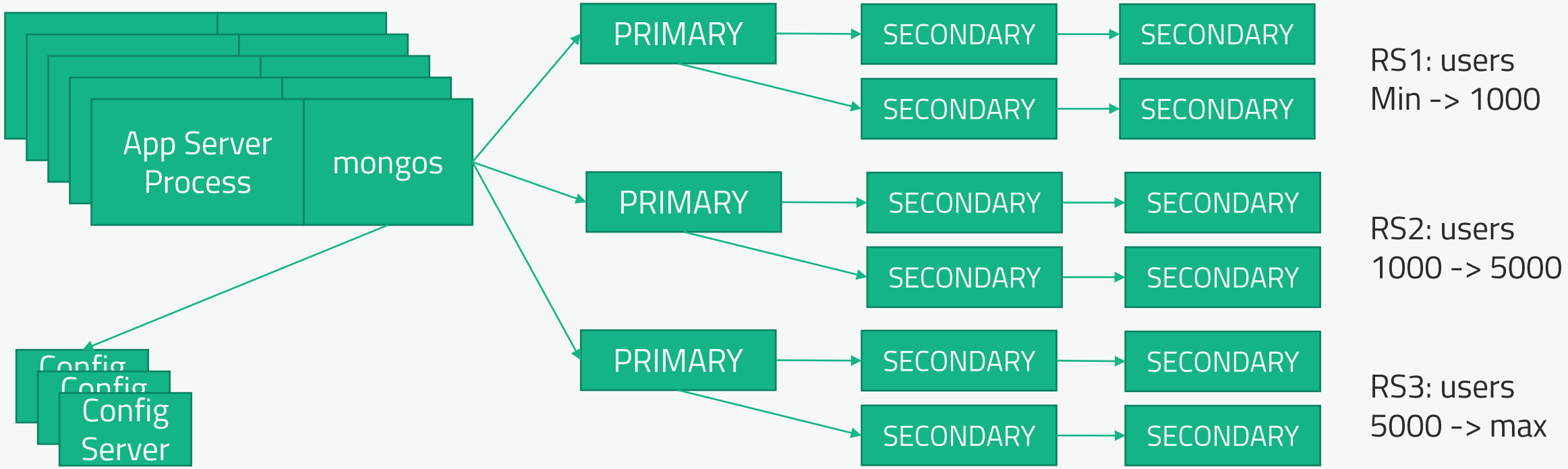
Failover servers: Hot Standby



Horizontal Scaling of Databases: Sharding

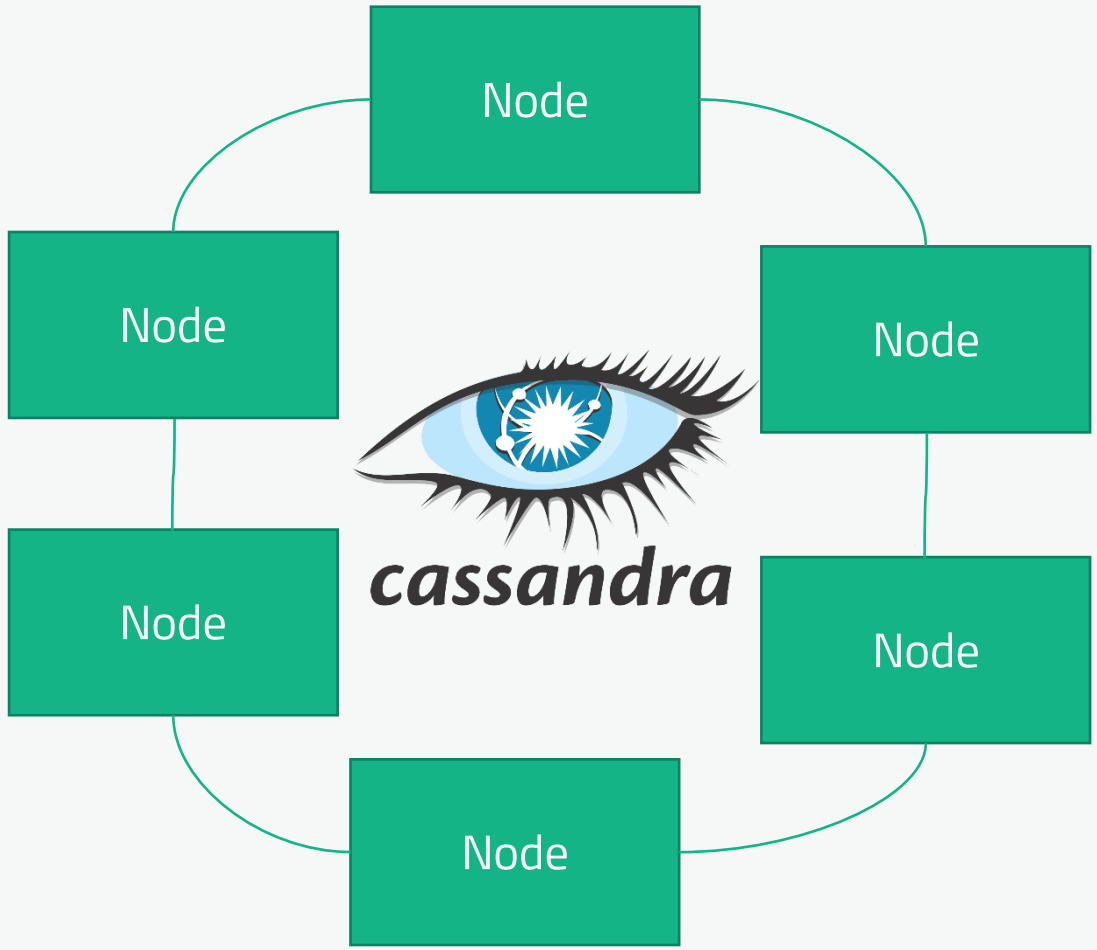


More Specific Example: MongoDB



RS = “Replica Set”

More Specific Example: Cassandra



Sharded databases are sometimes called “NoSQL”

- Tough to do joins across shards.
- Resharding
- Hotspots
- Most “NoSQL” databases actually do support most SQL operations and use SQL as their API.
- Still works best with simple key/value lookups.
- A formal schema may not be needed.
- Examples: MongoDB, DynamoDB, Cassandra, HBase

Denormalizing

NORMALIZED DATA: Less storage space, more lookups, updates in one place

Reservation ID	Customer ID	Time
1	123	6:30
2	451	7:00
3	123	8:00

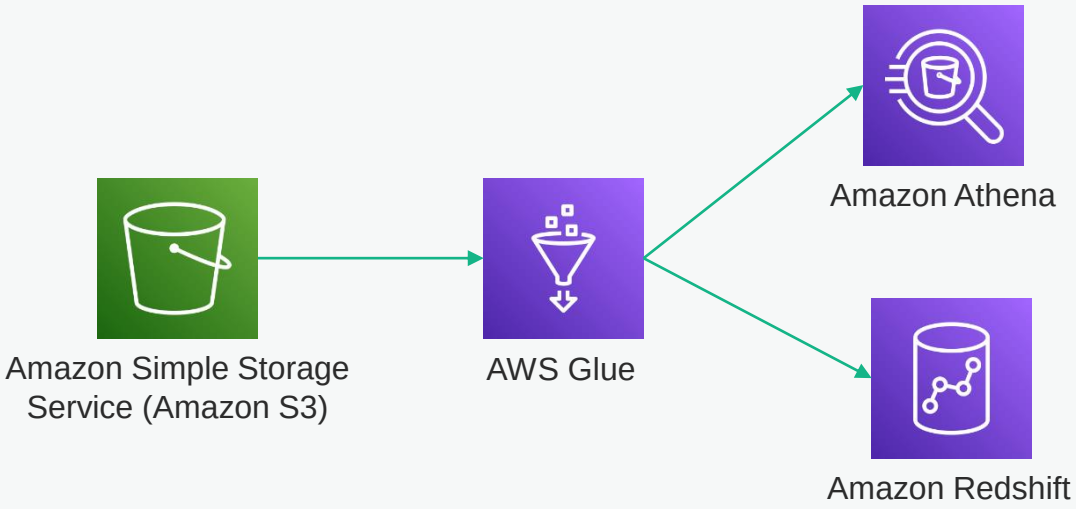
Customer ID	Name	Phone
123	Frank Kane	555-1234
451	John Smith	555-5233

DENORMALIZED DATA: More storage place, one lookup, updates are hard

Reservation ID	Customer ID	Name	Phone	Time
1	123	Frank Kane	555-1234	6:30
2	451	John Smith	555-5233	7:00
3	123	Frank Kane	555-1234	8:00

Cloud Solutions / Data Lakes

- Another approach is to just throw data into text files (csv, json perhaps) into a big distributed storage system like Amazon S3
  - This is called a “data lake”
  - Common approach for “big data” and unstructured data
- Another process (i.e., Amazon Glue) creates a schema for that data
- And cloud-based features let you query the data.
  - Amazon Athena (serverless)
  - Amazon Redshift (distributed data warehouse)
- You still need to think about how to partition the raw data for best performance.



ACID Compliance

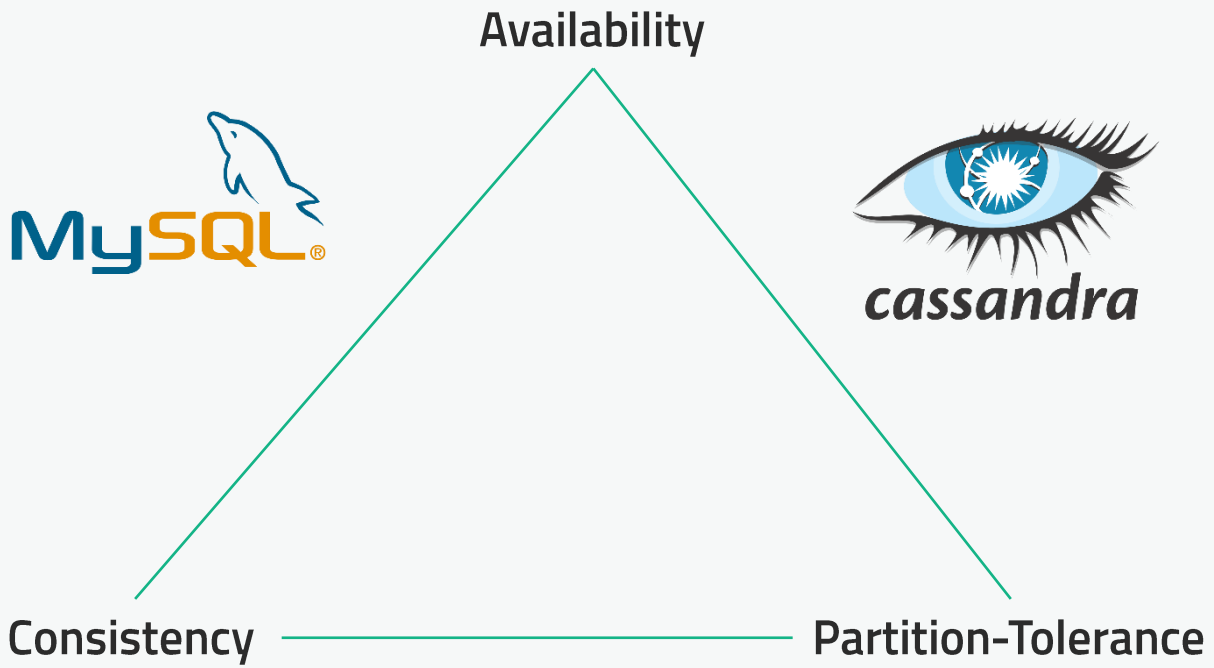
Atomicity: Either the entire transaction succeeds, or the entire thing fails.

Consistency: All database rules are enforced, or the entire transaction is rolled back.

Isolation: No transaction is affected by any other transaction that is still in progress.

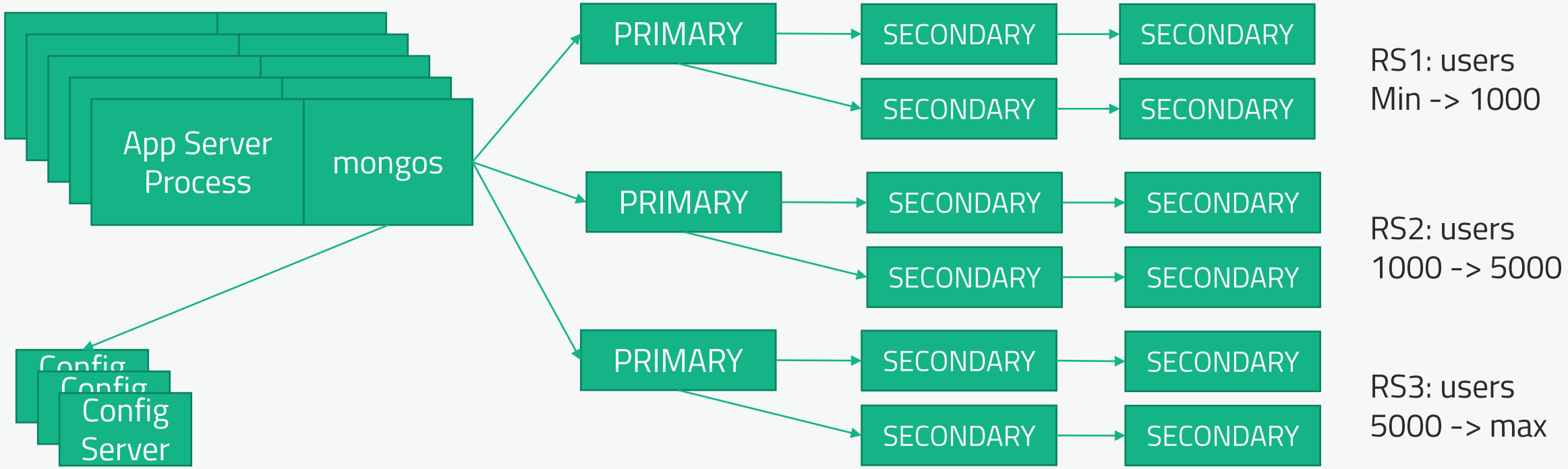
Durability: Once a transaction is committed, it stays, even if the system crashes immediately after.

The CAP Theorem



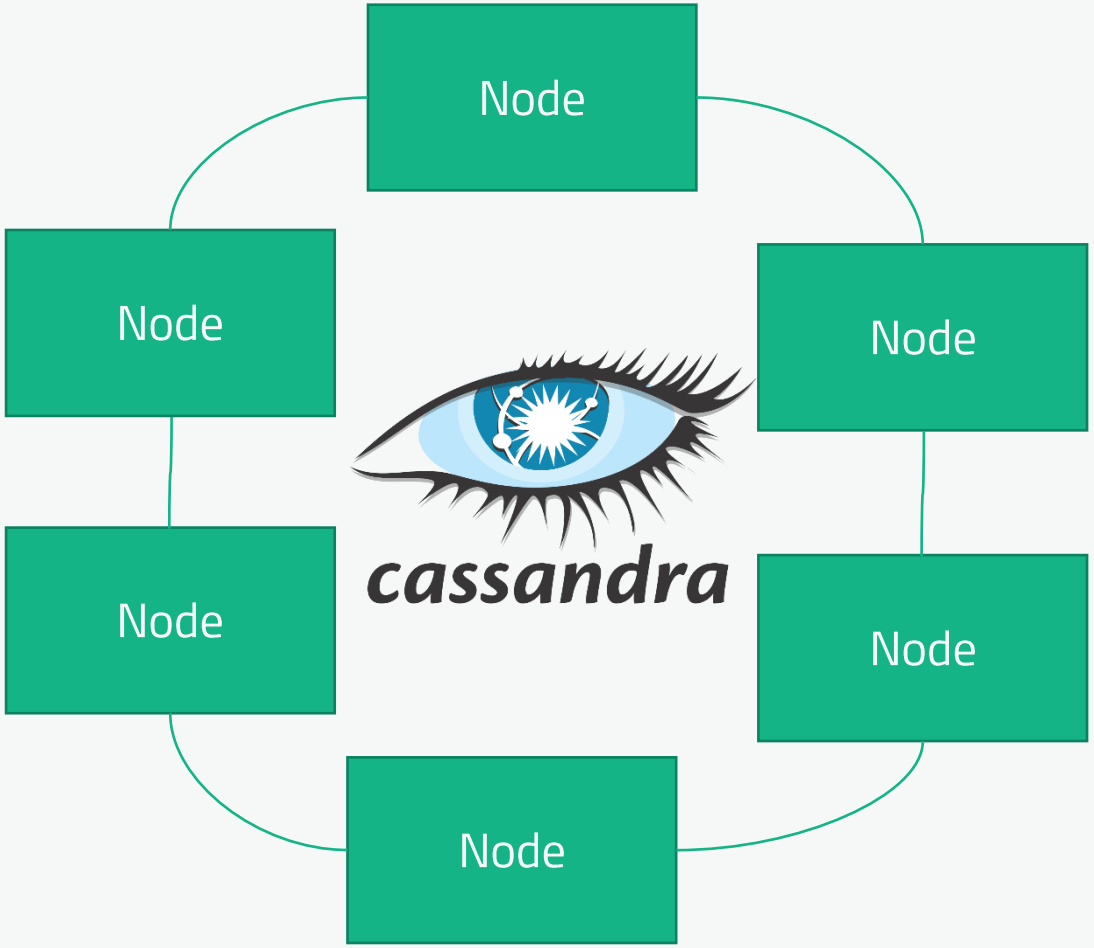
(In strongly consistent mode)

MongoDB: Single Master; trades off availability



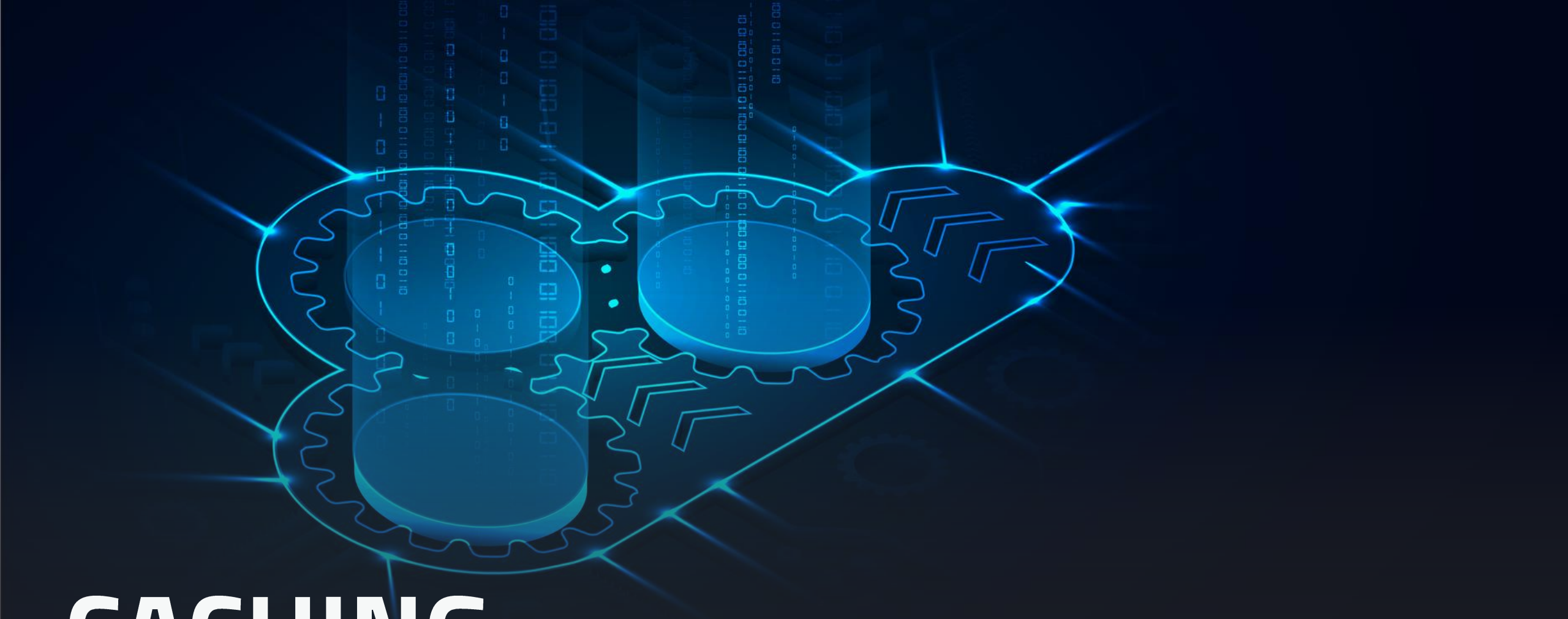
RS = “Replica Set”

Cassandra: No single master, eventually consistent



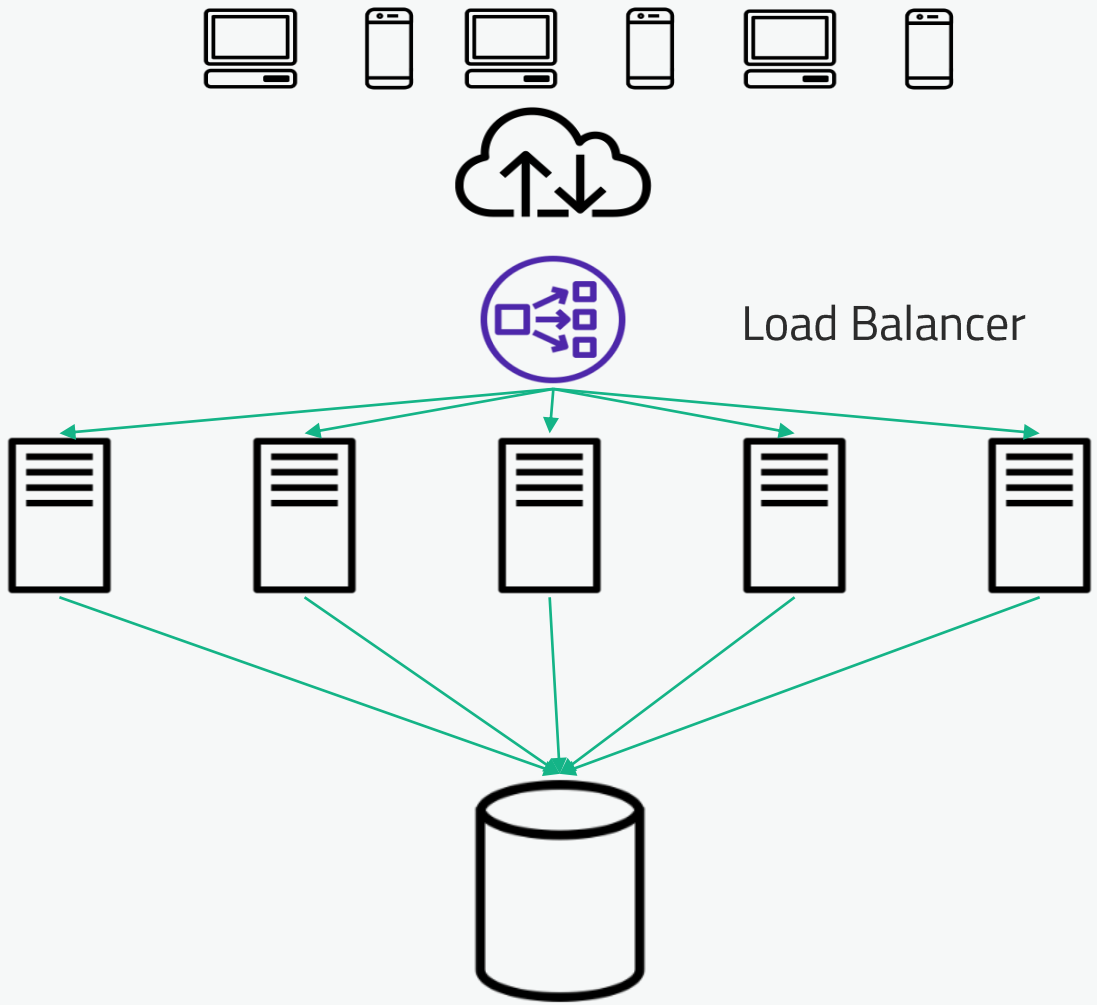
Be sure to understand requirements about scale, consistency, and availability before proposing a specific database solution

ASK QUESTIONS

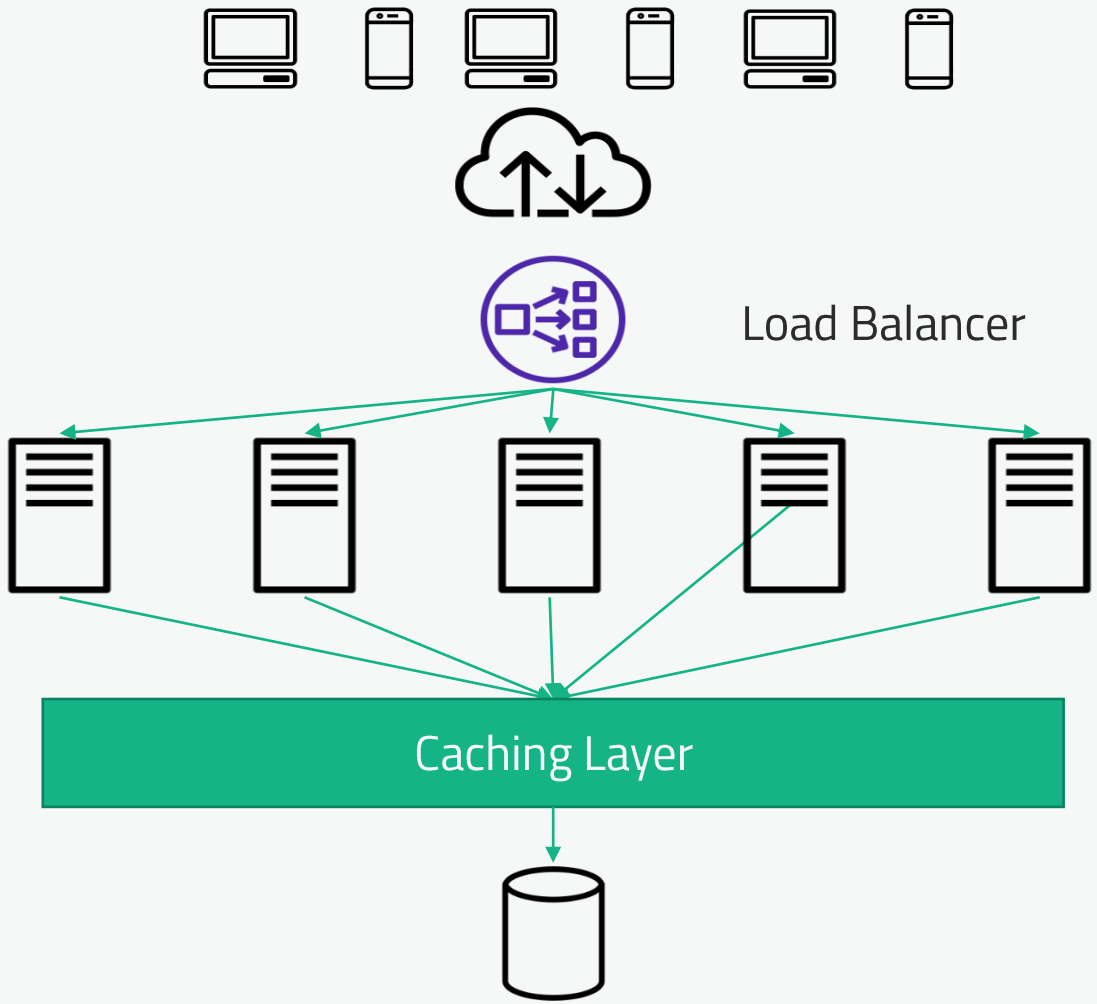


CACHING

Caching Layers

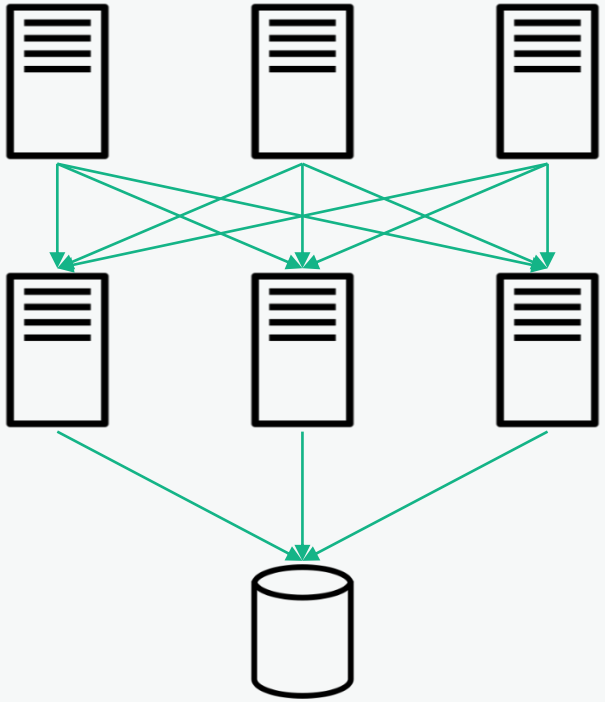


Caching Layers



How Caches Work

- Horizontally scaled servers
- Clients *hash* requests to a given server
- In-memory (fast)
- Appropriate for applications with more reads than writes
- The *expiration policy* dictates how long data is cached. Too long and your data may go stale; too short and the cache won't do much good.
- *Hotspots* can be a problem (the "celebrity problem")
- Cold-start is also a problem. How do you initially warm up the cache without bringing down whatever you are caching?

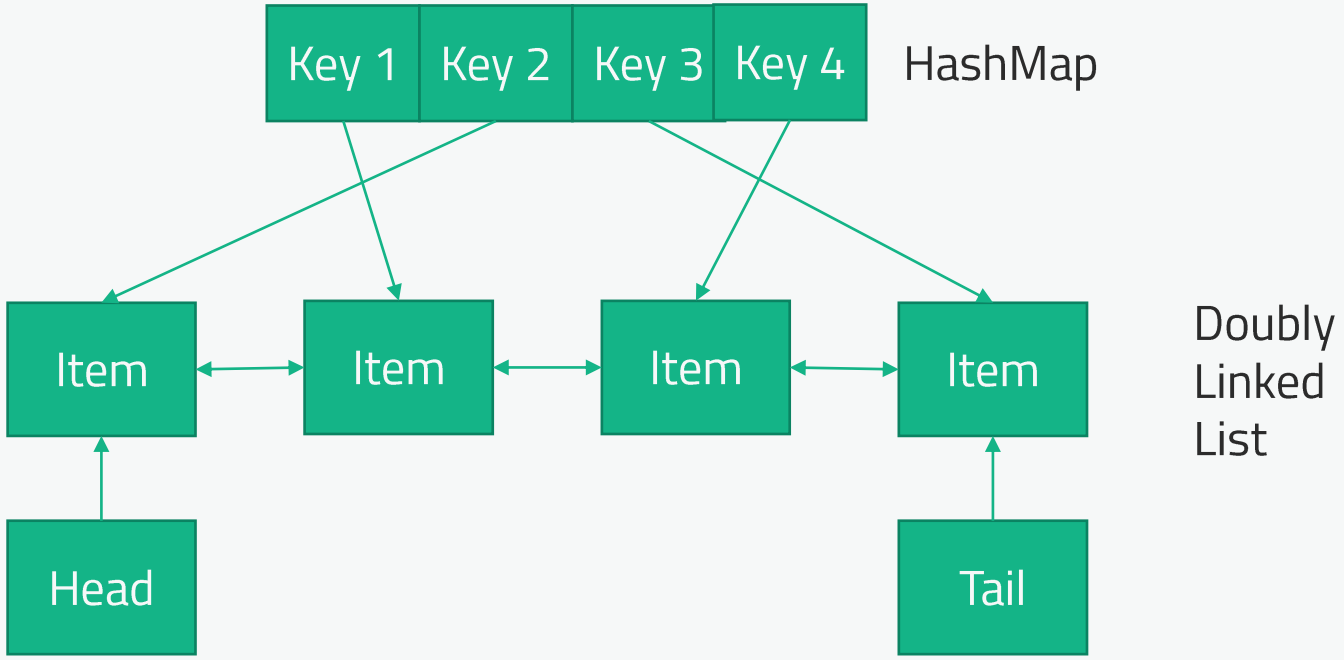


App / web servers

Cache servers

Eviction Policies

- LRU: Least Recently Used
- LFU: Least Frequently Used
- FIFO: First In First Out



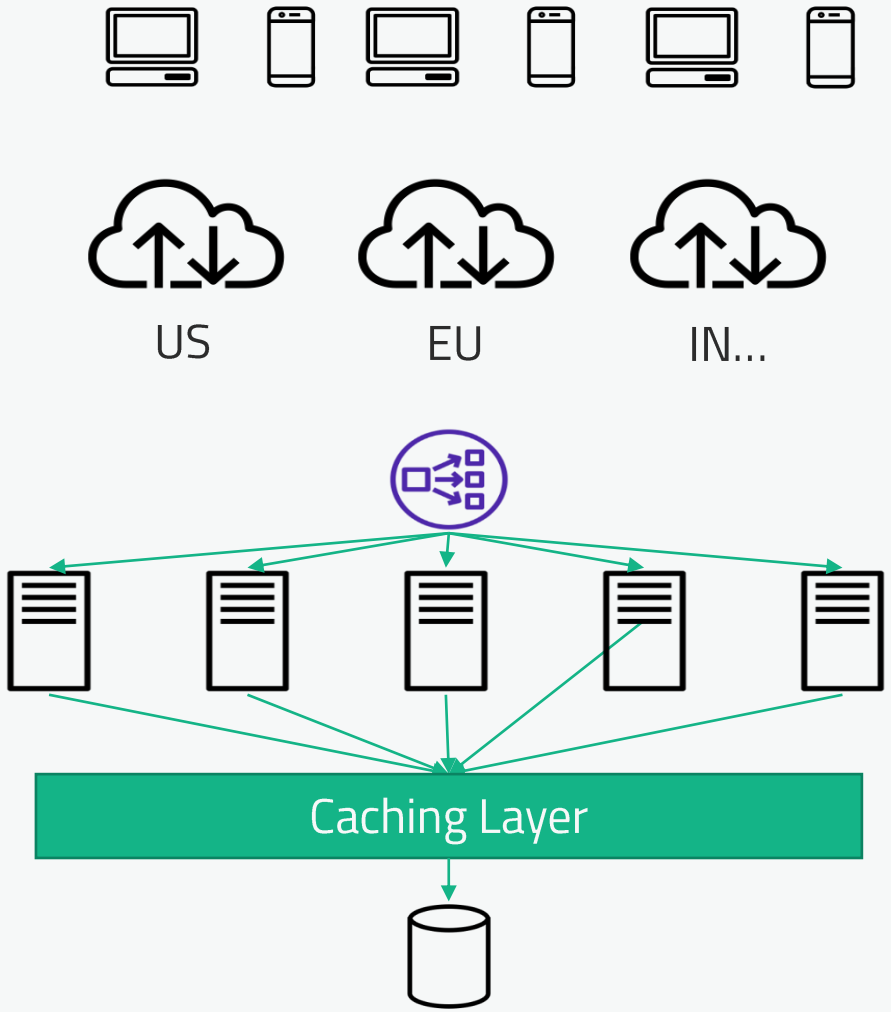
Sample LRU data architecture
(for a given shard)

A Few Caching Technologies

Memcached	Redis	Ncache	Ehcache	ElastiCache
▪ In-memory key/value store ▪ Open source.	▪ Adds more features ▪ Snapshots, replication, transactions, pub/sub ▪ Advanced data structures ▪ More complex in general	▪ Made for .NET, Java, Node.js	▪ Java ▪ Just a distributed Map really	▪ Amazon Web Services (AWS) solution ▪ Fully-managed Redis or Memcached

Content Delivery Networks (CDNs)

- Geographically distributed
- Local hosting of
  - HTML
  - CSS
  - Javascript
  - Images
- Some limited computation may be available as well
- Mainly useful for static content, such as images or static web pages.
  - You probably won't be asked to design a static web page, though!



CDN Providers

AWS CloudFront

Google Cloud CDN

Microsoft Azure CDN

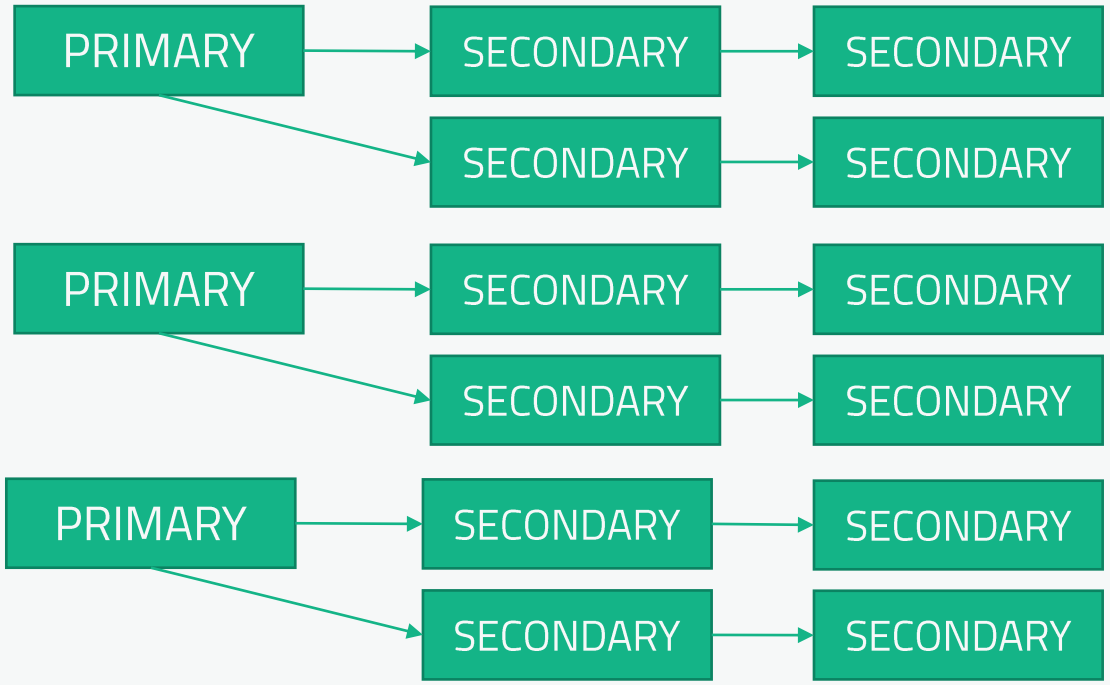
Akamai

CloudFlare

...and many more

RESILIENCY

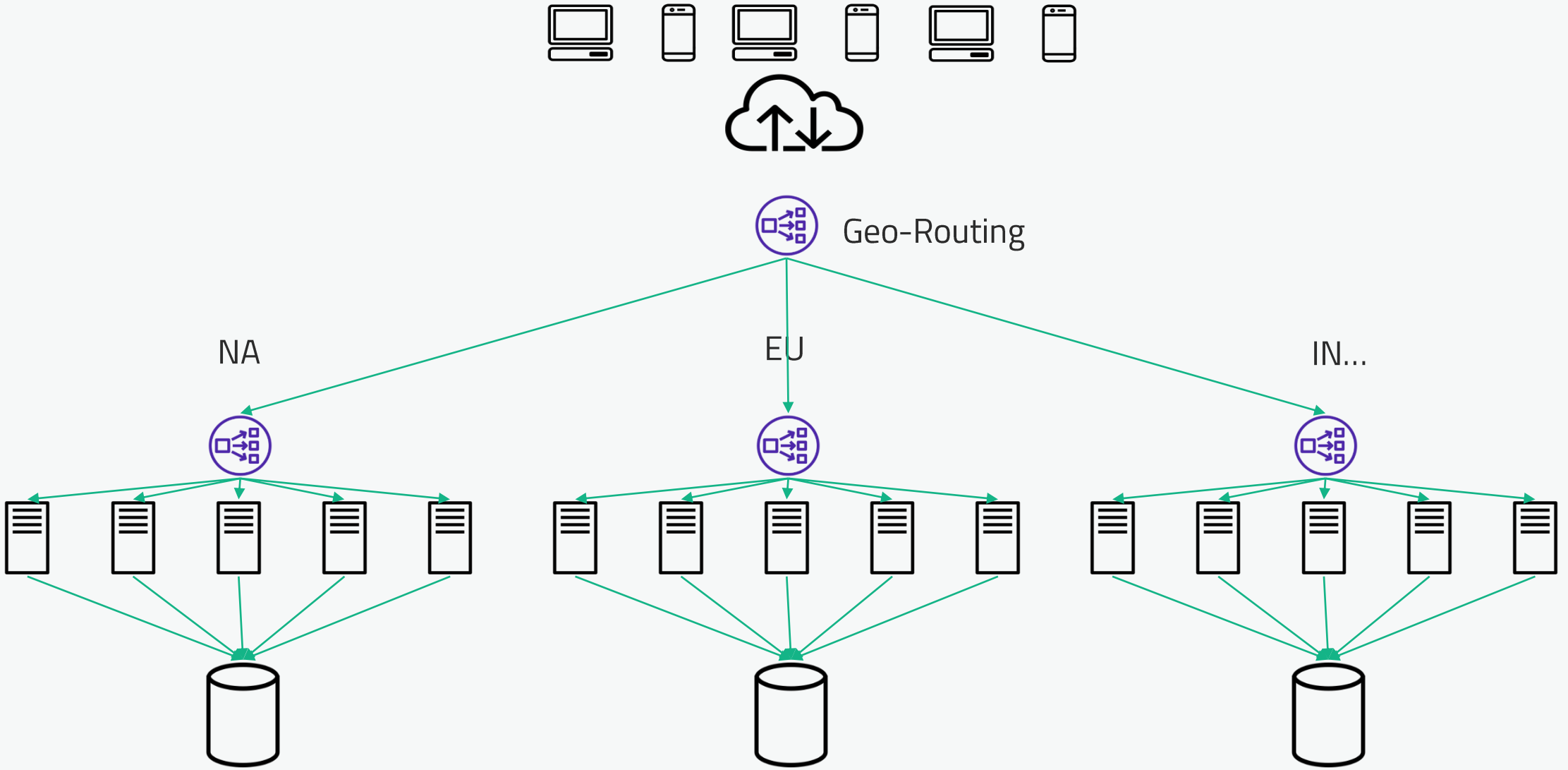
We've already talked about backup hosts



...but what about a real disaster?

Things that can fail

- A single server
- An entire rack
- An entire data center (AKA “availability zone”)
- An entire region
- …anything more, and you have bigger problems…


Geo-Routing
NA
EU
IN...

Be smart about distributing your servers

- Secondaries should be spread across multiple racks, availability zones, and regions
- Make sure your system has enough capacity to survive a failure at any reasonable scale
    - This means overprovisioning
- You may need to balance budget vs. availability. Not every system warrants this.
    - Provisioning a new server from an offsite backup might be good enough.
    - Again, ask questions!

DISTRIBUTED STORAGE

Distributed storage solutions

- Services for scalable, available, secure, fast object storage
- Use cases: “data lakes”, websites, backups, “big data”
- Highly durable:
- Amazon S3 offers 99.999999999% durability!

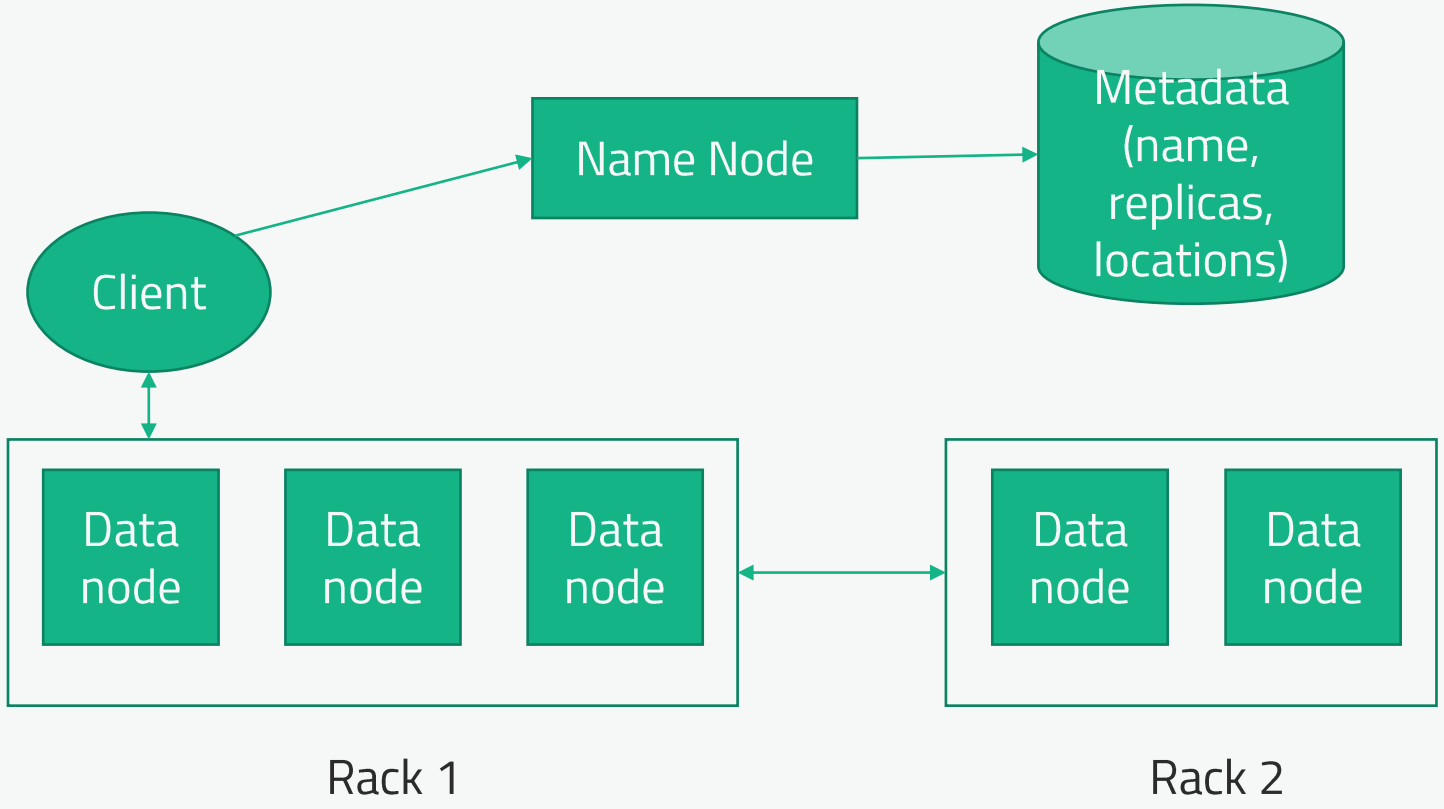
A brief diversion about SLA's

- What do we mean by 99.999999999% durability?
  - This is a *percentile*
  - This one is "11 9's" of durability
  - Meaning: there is a 0.000000001% chance of losing your data with S3.
- This can also be applied to *latency*, or how quickly a service responds to a request.
  - For example: you can say your "3 nines" latency is 100ms, meaning that 99.9% of requests come back within 100ms.
- Availability SLA's can be deceiving…
  - 99% availability would still result in 3.65 DAYS of downtime in a year
  - Whereas 99.9999% (6 nines) would result in about 30 seconds of downtime

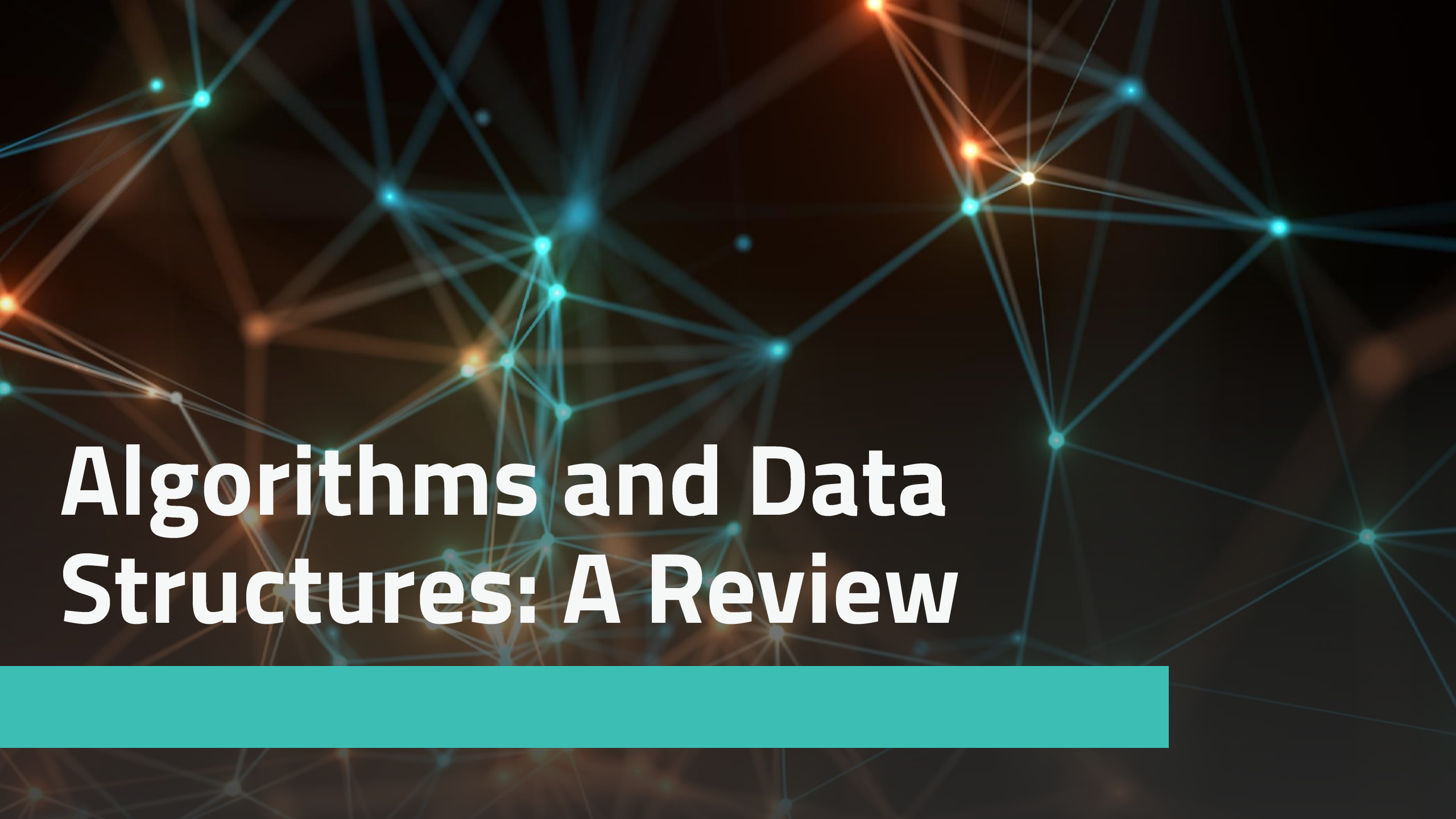
Distributed storage solutions

- Amazon S3
  - Pay as you go
  - Different tiers, ie Glacier for archiving is cheaper, but harder to read from. You can also choose the amount of redundancy you need to save money.
  - Hot / cool / cold storage
- Google Cloud Storage
- Microsoft Azure
- Hadoop HDFS
  - Typically self-hosted
- Then there are all the consumer-oriented storage solutions
  - Dropbox, Box, Google Drive, iCloud, OneDrive, etc.
  - Generally not relevant to system design

Example: HDFS architecture

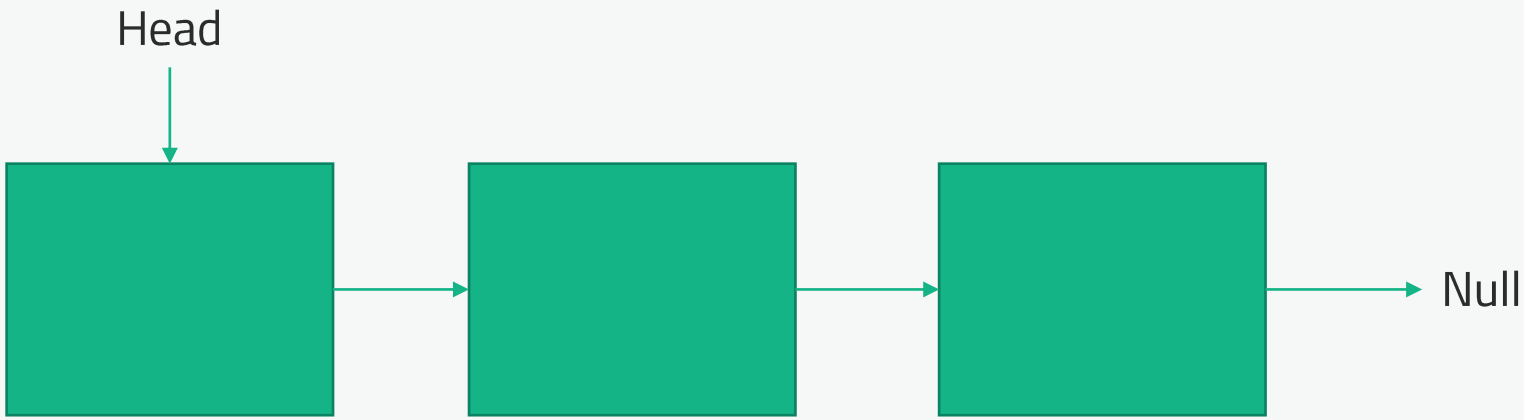


- Files are broken up into "blocks" replicated across your cluster
- Replication is rack-aware
- A master "name node" coordinates all operations
- Clients try to read from nearest replica
- Writes get replicated across different racks
- For high availability, there may be 3 or more name nodes to fall back on, and a highly available data store for metadata



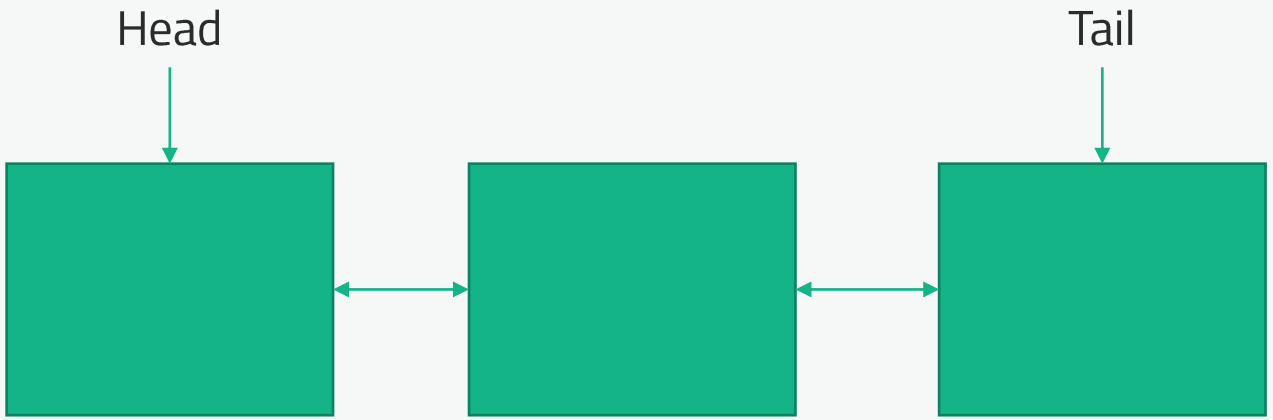
Algorithms and Data Structures: A Review

Singly Linked List



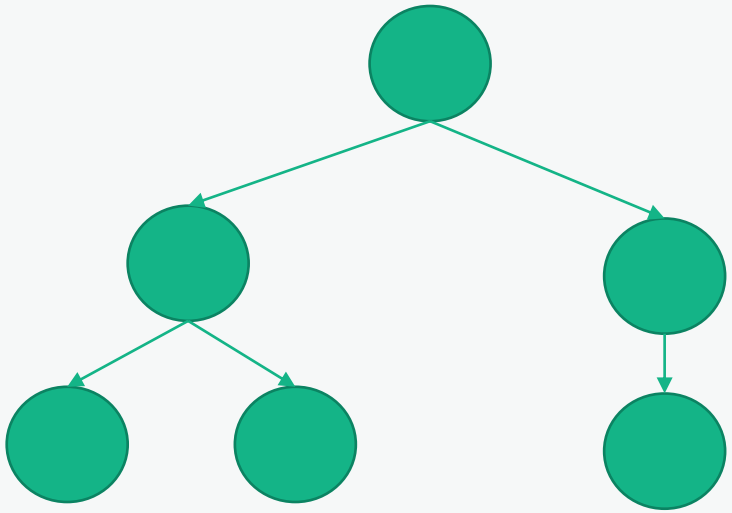
- Grows dynamically (as opposed to an array)
- Access is O(n)
- Inserts at head is O(1)
- Insert at end is O(n)
- Best for use cases that involve sequential access
- Also good for stacks (LIFO), or queues (FIFO) if you keep track of the tail as well
- One pointer per node = low memory requirements

Doubly Linked List



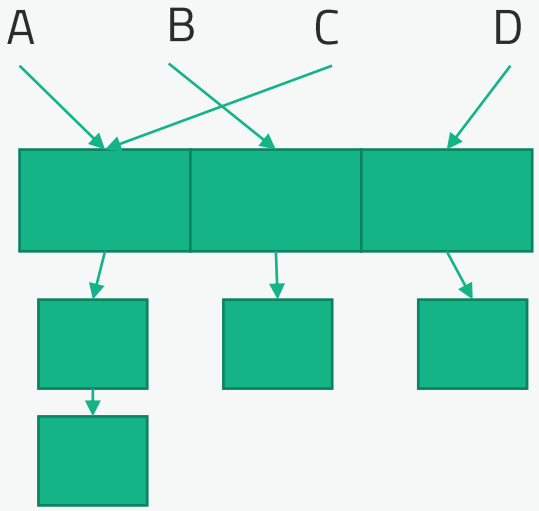
- Each node has a “next” and “previous” pointer
- Insert at front or back is O(1)
- Access is still O(n) (but could be faster in practice, since you can start at either end)
- Useful for Deques
- MRU: always move most recent access to the head

Binary Tree



- Each element has a left and right child
- If the left and right are ordered (i.e., left means “less than”) it is a binary search tree
- Access is O(log(n)) on average, O(n) worst case
- Insert / delete also O(log(n)) as you need to do another search to rearrange things
- Mostly used in cases where you need to do in-order traversals

Hash Table



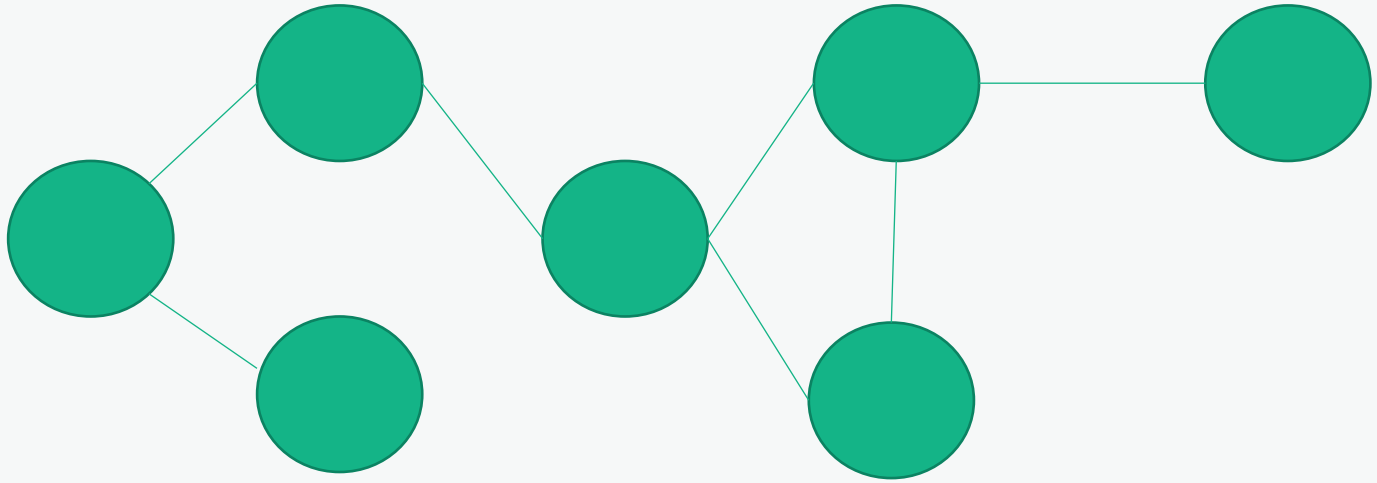
Hash function

Buckets

Lists (or something)

- A “hash function” quickly maps some *key* to a *bucket*
- That bucket is then searched for the key’s *value*
- *Hash collisions* occur when more than one key maps to the same bucket
- Inserts, lookups and deletions are O(1)… but O(n) in the worst case.
- Used when fast lookups are needed

Graph

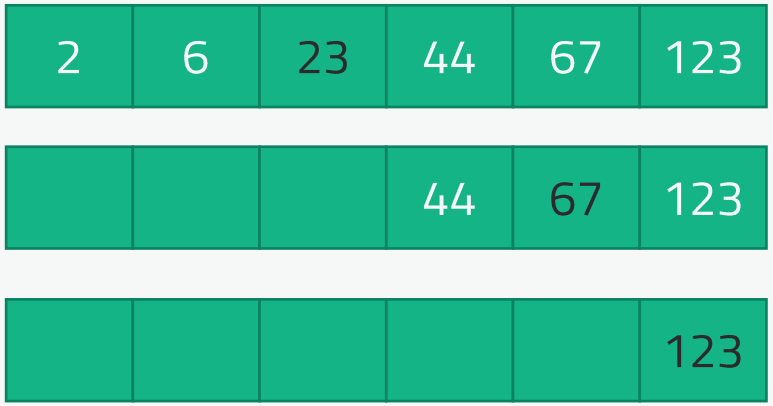


- Consists of nodes (vertices) than can be connected in arbitrary ways (edges)
- For example, friends in a social network, paths in a city, networks in general.
- Traversal strategies include Breadth-First-Search (BFS) and Depth-First-Search (DFS)
- Access is O(V+E)

Linear Search

- Start with an array (or list)
- This example is sorted but it doesn't have to be
- Start at the beginning and keep going until you find what you're looking for.
- O(n)

Binary Search



- Start with a sorted array (or list)
- Start at the middle, split the array in 2
- If what you're looking for is bigger, move to the second half of the array
  - Or if it's smaller, move the first half.
- Check the middle, split the half you're looking at again if necessary
- Repeat until you find it
- O(log(n))

Sorting Algorithms

- Unlikely to come up in the context of system design, so a quick review
- Insertion sort: O(n) best case, $O(n^2)$ worst case
  - OK for small or mostly-sorted lists
- Merge Sort: O(n log(n))
  - Scales well to large lists
- Quicksort: O(n log(n))
  - Very fast, unless you hit the worst case scenario of $O(n^2)$ due to poor choice of a pivot point
  - Some complex implementations to avoid that
- Bubble sort: $O(n^2)$
  - Simple but inefficient
- Many others…

Search and Information Retrieval

General recipe:

- Start with a *forward index* of keywords in each document
  - i.e., Document ID 123 => "the","quick","red","fox"
  - Problems: capitalization, spaces, punctuation, offensive terms, phrases
  - Other signals of relevance can be included in addition to position (formatting, etc)
- Then generate an *inverted index* that maps keywords to documents
- Somehow those documents need to be ranked
  - Could just be a function of how often the keyword appears and where

Keyword	Document ID, Position tuples
"Palm tree"	(432,1), (36,1235),(432,55)
"Dinosaur"	(22,2), (22,253),(724,4342),(552,793)

Keyword	Document IDs
"Palm tree"	432,36
"Dinosaur"	22,552,724

TF-IDF: Document search

- Stands for *Term Frequency* and *Inverse Document Frequency*
- Important data for search – figures out what terms are most relevant for a document
- Sounds fancy!
  - But it's really one of the oldest and most basic search algorithms.

TF-IDF Explained

- *Term Frequency* just measures how often a word occurs in a document
  - A word that occurs frequently is probably important to that document's meaning
- *Document Frequency* is how often a word occurs in an entire set of documents, i.e., all of Wikipedia or every web page
  - This tells us about common words that just appear everywhere no matter what the topic, like "a", "the", "and", etc.

- So a measure of the relevancy of a word to a document might be:

$$\frac{Term\ Frequency}{Document\ Frequency}$$

Or: Term Frequency * Inverse Document Frequency

That is, take how often the word appears in a document, over how often it just appears everywhere. That gives you a measure of how important and unique this word is for this document

Applying TF-IDF to Search

- A very simple search algorithm could be:
  - Compute TF-IDF for every word in a corpus
  - For a given search word, sort the documents by their TF-IDF score for that word
  - Display the results
- Note computing “document frequency” can be an intractable problem if we’re talking about the entire web.

PageRank: Searching the Web

- Google's original trick. Inspired by citations of academic papers.
- Instead of relying entirely on the contents of a page, also look at the page's backlinks and the anchor text for those links.
- A page with lots of inbound links means it might be more useful.
- The anchor text on those links are treated like additional keywords for the page.
- A given backlink is weighted by how many other links are on the page it's coming from
- A dampening factor means we don't follow links forever without losing weight on them.
- Today Google has moved way past things like PageRank and TF/IDF. Deep learning is said to play a big role in ranking for example.

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right)$$

MESSAGE QUEUES

Message Queues as a scaling tool

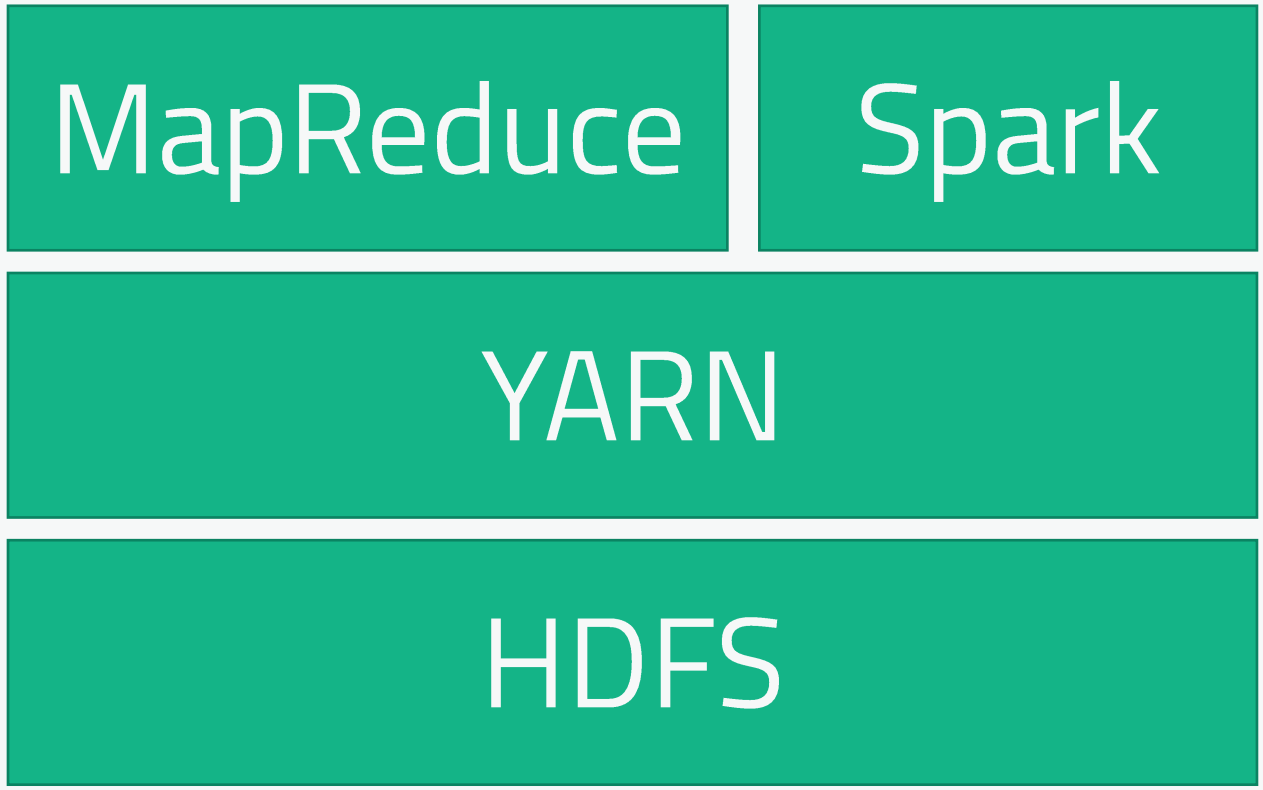
Publishers / Producers

Subscribed consumers

- Decouples producers & consumers
- So if the consumers get backed up, that's OK.
- Example: Amazon SQS service
  - Single-consumer vs. pub/sub
- This is different from *streaming* data (generally real-time, massive data)

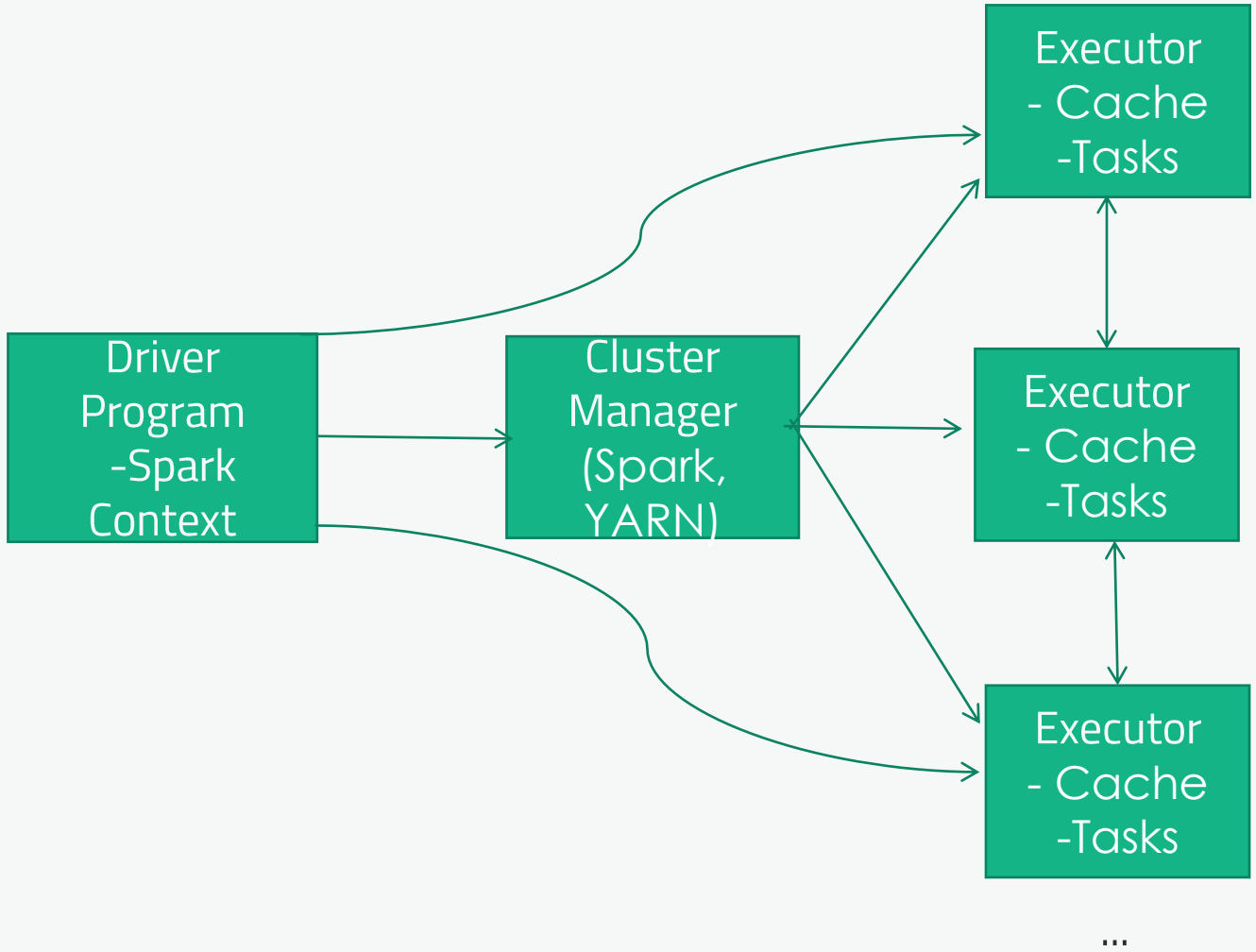

APACHE SPARK

Apache Spark



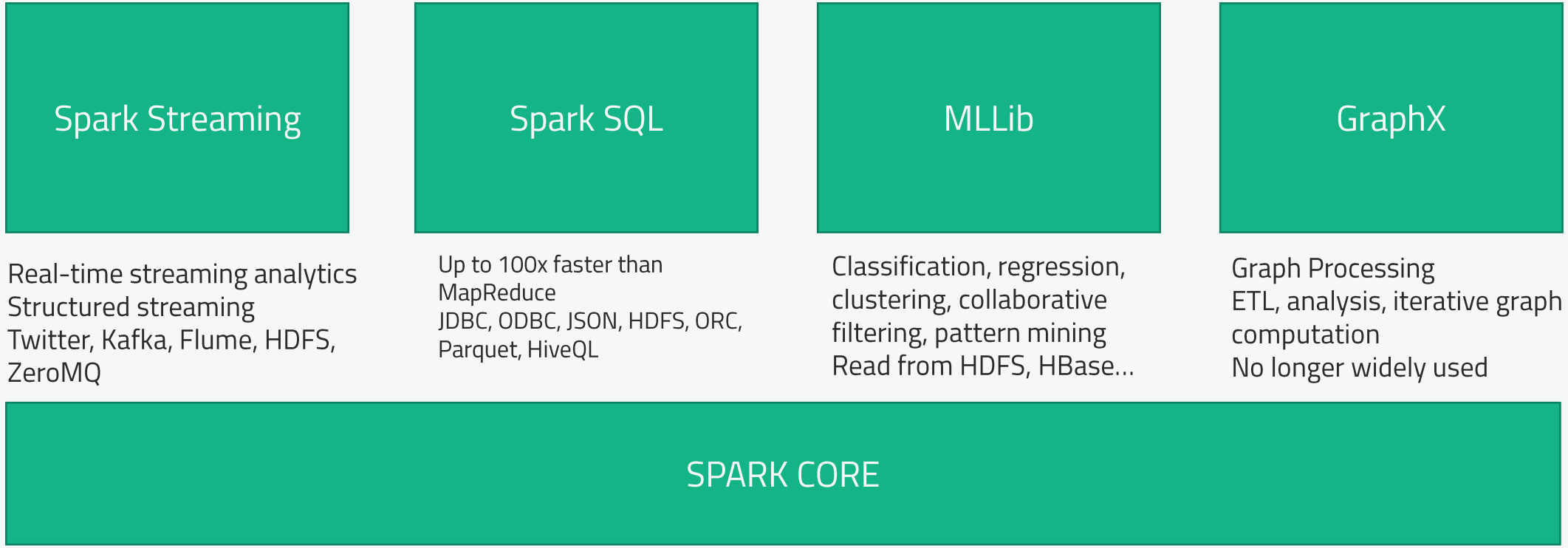
- Distributed processing framework for big data
- In-memory caching, optimized query execution
- Supports Java, Scala, Python, and R
- Supports code reuse across
  - Batch processing
  - Interactive Queries
    - Spark SQL
  - Real-time Analytics
  - Machine Learning
    - MLLib
  - Graph Processing
- Spark Streaming
  - Integrated with Kinesis, Kafka, on EMR
- Spark is NOT meant for OLTP

How Spark Works



- Spark apps are run as independent processes on a cluster
- The SparkContext (driver program) coordinates them
- SparkContext works through a Cluster Manager
- Executors run computations and store data
- SparkContext sends application code and tasks to executors

Spark Components

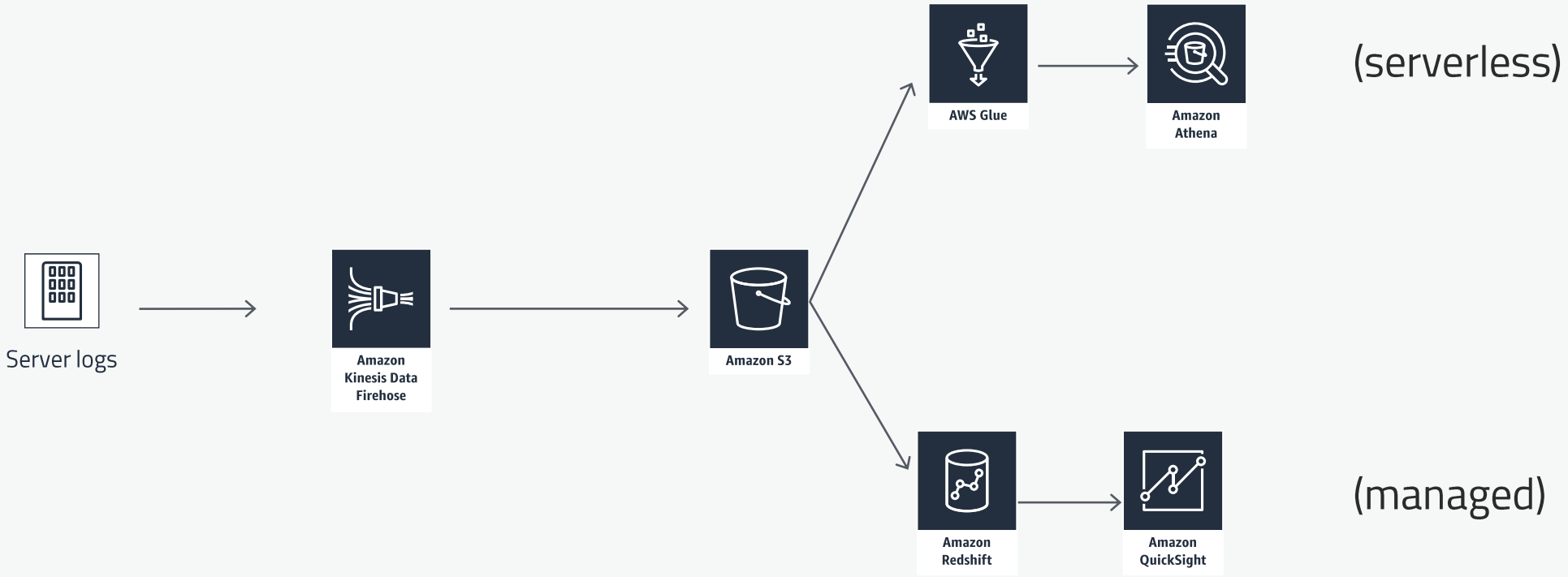


Memory management, fault recovery, scheduling, distribute & monitor jobs, interact with storage
Scala, Python, Java, R

CLOUD COMPUTING:
a very brief review

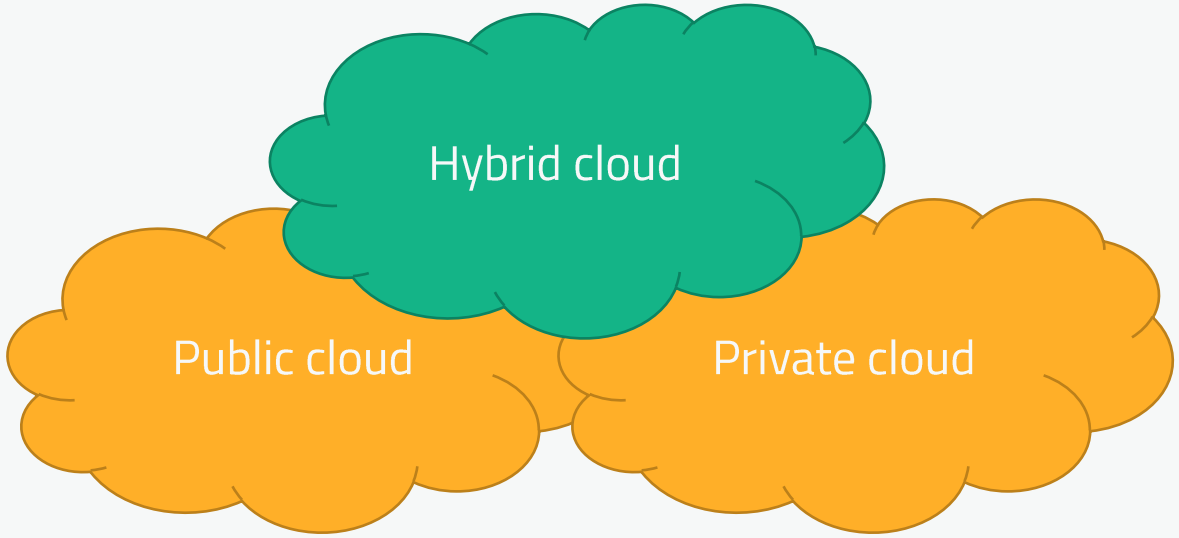
	Amazon Web Services (AWS)	Google Cloud	Microsoft Azure
Storage	S3	Cloud Storage	Disk, Blob, or Data Lake Storage
Compute	EC2	Compute Engine	Virtual Machines
NoSQL	DynamoDB	Bigtable	CosmosDB / Table Storage
Containers	Kubernetes / ECR / ECS	Kubernetes	Kubernetes
Data streams	Kinesis	DataFlow	Stream Analytics
Spark / Hadoop	EMR	Dataproc	Databricks
Data warehouse	Redshift	BigQuery	Azure SQL / Database
Caching	ElastiCache (Redis)	Memorystore (Redis or Memcached)	Redis

Example: Design a Data Warehouse for Log Data with AWS



Hybrid Cloud

- Combine your own data centers ("on-premises" or "private cloud") with a public cloud (AWS, Google, Azure, etc.)
- Allows easy scaling of on-premises systems
- Allows for regulations that require certain data to be on-premises
- Requires bridges between your data center and the cloud
  - The specifics vary by cloud provider
- "Multi-Cloud" – more than one public cloud provider



INTERVIEW STRATEGIES

Start by Clarifying Requirements

You will be given some incredibly vague problem, like “Design YouTube”. It is up to you to turn this into concrete requirements your system must meet.

Start by repeating the question and confirm you understand it with the interviewer.

ASK LOTS OF QUESTIONS

THINK OUT LOUD

Working Backwards

- Start from the customer experience to define your requirements
  - (This will gain MAJOR POINTS at Amazon, but works in general.)
- YouTube Example:
  - How will users discover videos? Do we need to think about building a search engine? A recommender engine? An advertising engine?
  - Use this to limit the scope of what you're being asked to do.
  - Understand the *customer experience* you are being asked to deliver.

Working Backwards

- Identify WHO are the customers
- WHAT are their use cases
- WHICH use cases do you need to concern yourself with
  - You're not going to design all of YouTube in 20 minutes.
- Your initial task is to CLARIFY THE REQUIREMENTS of what you are designing.
- Your interviewer wants to see that you can think about problems from a business perspective and not a purely technical one.

Defining scaling requirements

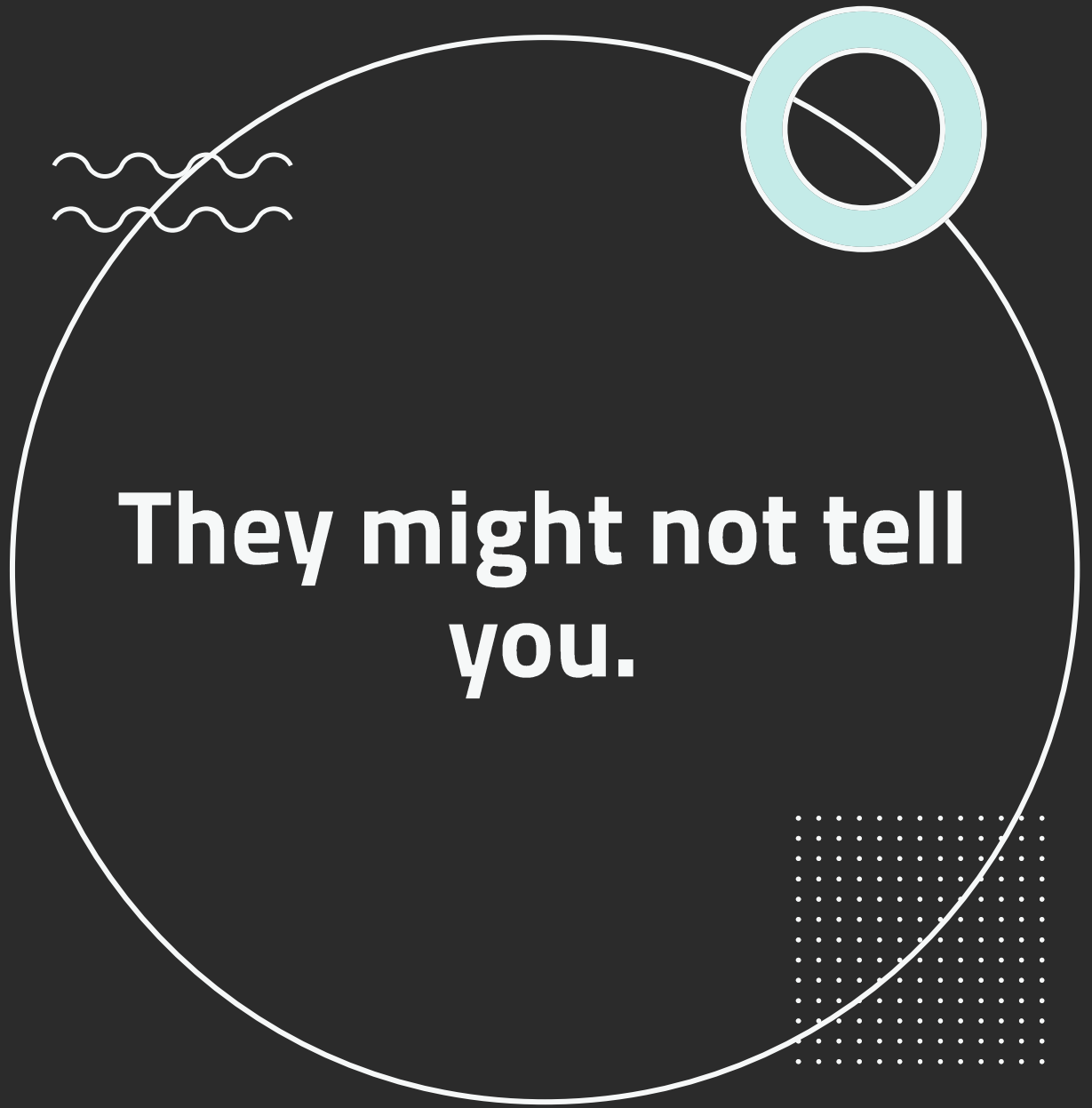
- Nail down the scale of the system. Is it hundreds of users? Millions?
  - This will inform you on the need for horizontal partitioning
  - How often are users coming? What *transaction rate* do you need to support?
- Also define the scale of the data.
  - Hundreds of videos? Millions?
- YouTube example: millions of users, millions of videos.
  - You will need every trick in the book for horizontally scaled servers and data storage.
- Some internal tool might not need this level of complexity, however.
  - Always prefer the simplest solution that will work.
  - Vertical scaling still has its place.

Defining latency requirements

- How fast is fast enough?
  - This informs the need for caching and CDN usage
  - (Caching is also a tool for scaling, however – it reduces load on services & data stores)
  - Try to express this in SLA language (i.e., 100ms at three-nines for a given operation)
- YouTube example:
  - Caching video recommendations
  - Caching video metadata, descriptions, etc.

Defining availability requirements

- How much downtime can you tolerate?
  - Is being down a threat to the business? Or just an inconvenience?
  - If the former, you need to design for high availability
    - Opt for redundancy across many regions / racks / data centers rather for simplicity or frugality



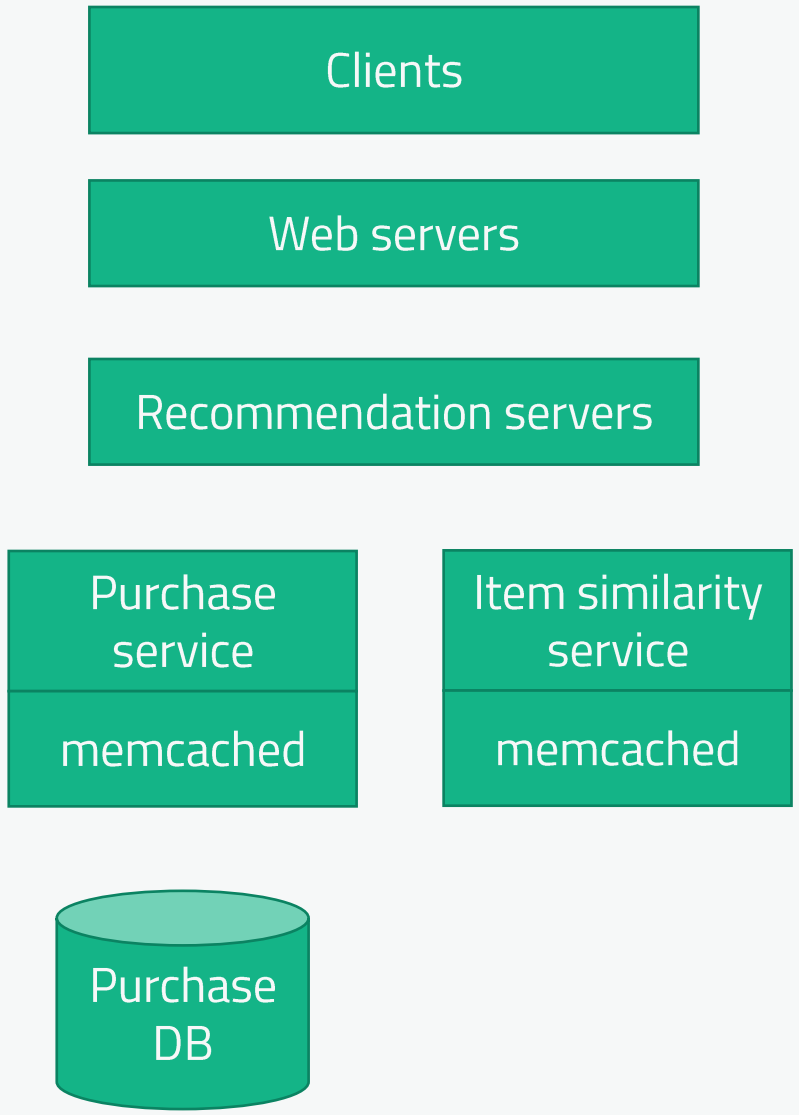
- Work backwards from the customer to estimate what sorts of requirements make sense from their standpoint.
- “Back of the envelope” calculations may be needed. (How many users and videos *does* YouTube have? You can make an educated guess.)
- Get buy-in from the interviewer before proceeding to design the system.

Think Out Loud

- Don’t just clam up for ten minutes while you think about things.
- Clarify requirements, define the constraints of what you need to build.
- Think out loud about the solutions you’re considering to meet those requirements
- Give the interviewer a chance to steer you in a different direction before you start diving into details
- You don’t know how much time you have for this part of the interview, so make every minute count.

Sketching Out Your Design

- Start with high-level components
- Work backwards if you can (especially at Amazon)
- Then flesh out each component as time permits
  - How do they scale?
  - How are they distributed for availability?
- Let the interviewer talk, listen to them. They may be trying to steer you in the right direction.
- Identify bottlenecks, maintenance, costs concerns as you go – show that you understand the tradeoffs of the choices you are making
- Notation and format generally doesn't matter much, as long as you can communicate what it means.



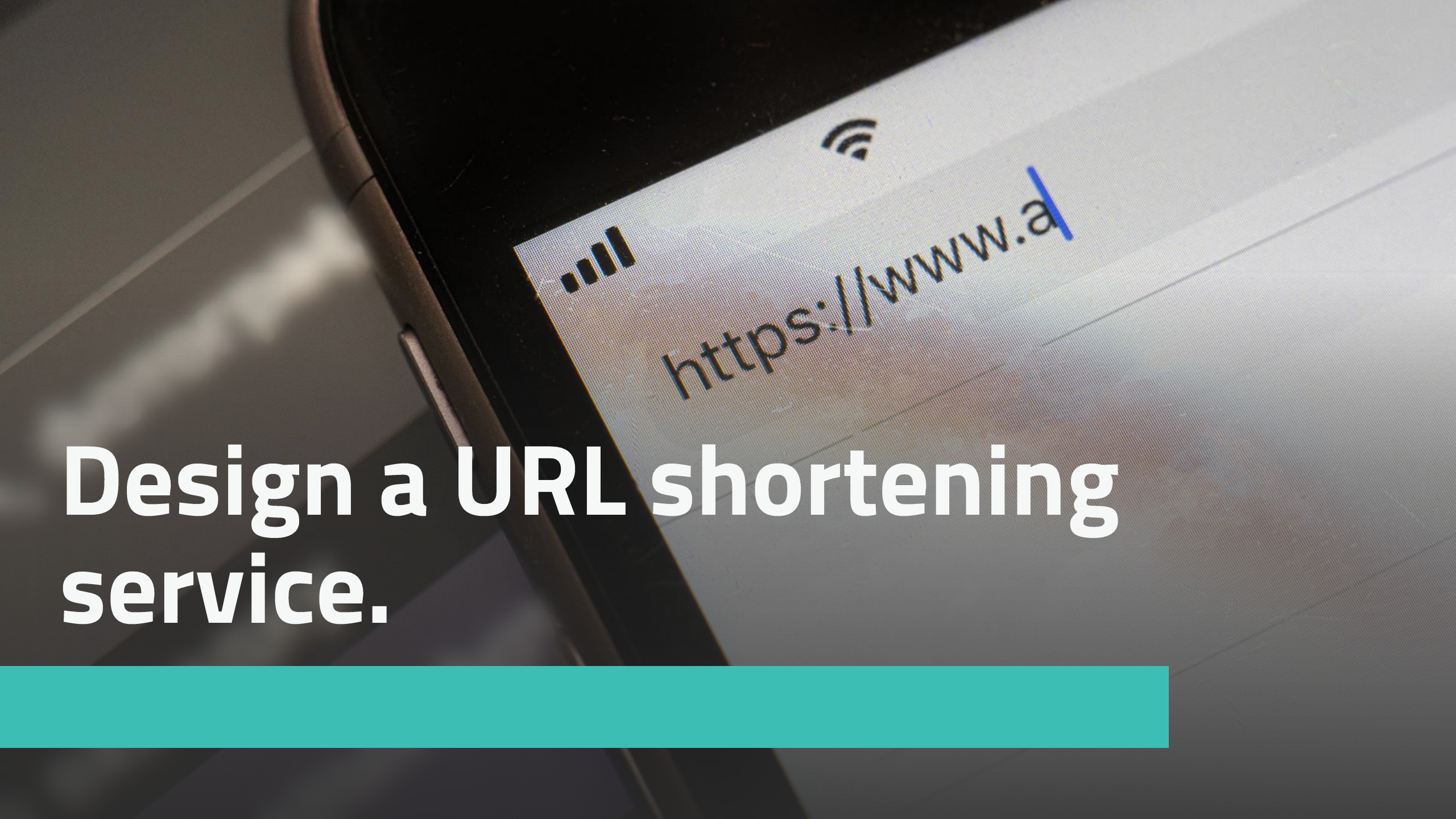
Be Honest

- Don’t pretend to know stuff you don’t know. That won’t end well.
- If you’re steered into a direction you’re unfamiliar with, say so.
- But don’t just give up! Try to think through it, working with the interviewer to come up with a solution collaboratively.
- This is an opportunity to demonstrate grit, perseverance, and the ability to work with others - which is more important than anything.

Defending Your Design

- The interviewer will try to poke holes in your design.
- What happens if X fails?
- What happens if we get a sudden surge of traffic / data?
- Did you meet the scaling & availability requirements you defined?
- Does your system meet all of the use cases discussed?
- How would you make it better?
- How would you optimize or simplify it?
- What is its operational burden? How will you monitor it?
- DON’T GET DEFENSIVE – take feedback constructively

Mock Interviews



Design a URL shortening service.

Design a URL shortening service. OK, so we're talking about something like bit.ly, right? A service where anyone can enter a URL, get a shorter URL to use in its place, and we manage redirecting them?

What sort of scale are we talking about?

Any restrictions on the characters we use? Symbols might be a little too hard for people to remember or type…

Well, how short is short?

a-z 0-9 = 36 characters

$$36^6 = 2176782336$$

How about vanity URL's? Can people specify their own URL if it's available?

Do we let them edit and delete short URL's once created?

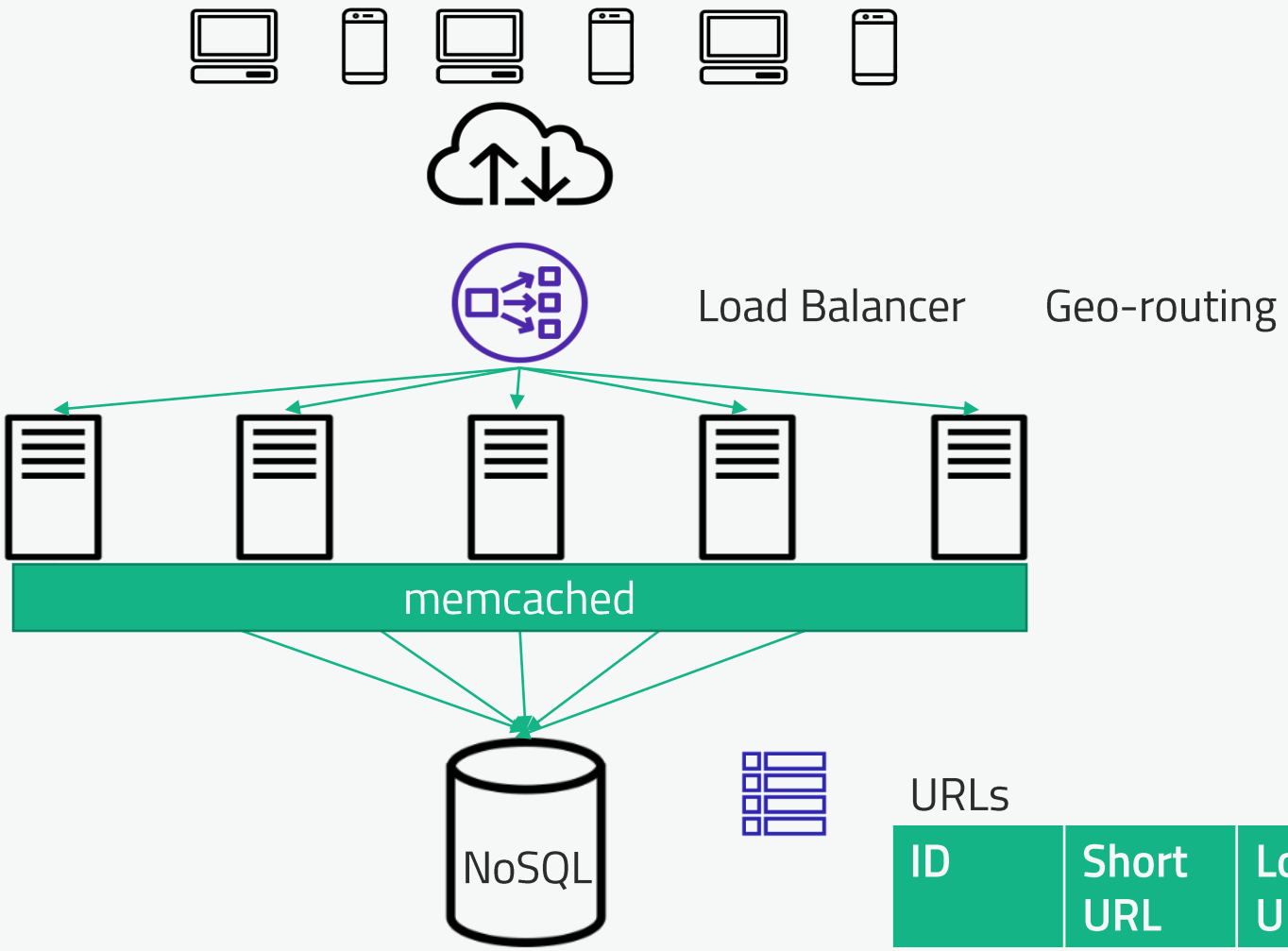
How long do shortened URL's last?

Try It Yourself!

- What API's would you need to implement this system?
- How does the redirection work at massive scale?

POST	Add new URL long URL, user ID (optional) -> status, short URL - What if someone else shortened it earlier?
POST	Add new vanity URL long URL, user ID, vanity URL -> status
PATCH	Update URL long URL, user ID, updated long URL -> status, existing short URL
DELETE	Delete URL long URL, userID -> status
GET	Display mapping long URL, userID -> status, short URL
GET	Redirect short URL -> redirect to long URL

GET /abc123



URLs

ID	Short URL	Long URL	userID

DEBRIEF

- Started by repeating the question and clarifying requirements
- Worked backwards from the customer experience
- The API was critical to the operation of the system, so we started by thinking about the specific operations we needed to support
- Proposed a horizontally scalable fleet of app servers, distributed to maximize availability
- Proposed an appropriate distributed database
- Did not get defensive when challenged by the interviewer
- We at least mentioned security, availability, and scaling concerns along the way.

Design a restaurant reservation system.

OK, you want me to design a restaurant reservation system. Is this just for one restaurant, or for any number of restaurants like OpenTable or something?

Alright, let's think about the user experience first. A user will want to select a restaurant, enter their party size, find a list of available times near the time they want, lock in their reservation, and get some sort of confirmation via SMS or something. They'll also need some way to change or cancel reservations.

So there are probably thousands of restaurants out there that might be a part of this system, and tens or hundreds of thousands of diners. They'll expect this system to be fast and reliable. Am I right in thinking we should optimize for performance and reliability over cost?

I suppose the restaurant is also a customer… what would they need? Reporting, analytics, a way to set up how many tables and their configurations, how many tables to hold aside for walk-ins, a way to contact reservation holders…

Let me sketch some thoughts on the data we'll need while I'm thinking of it…

Try It Yourself!

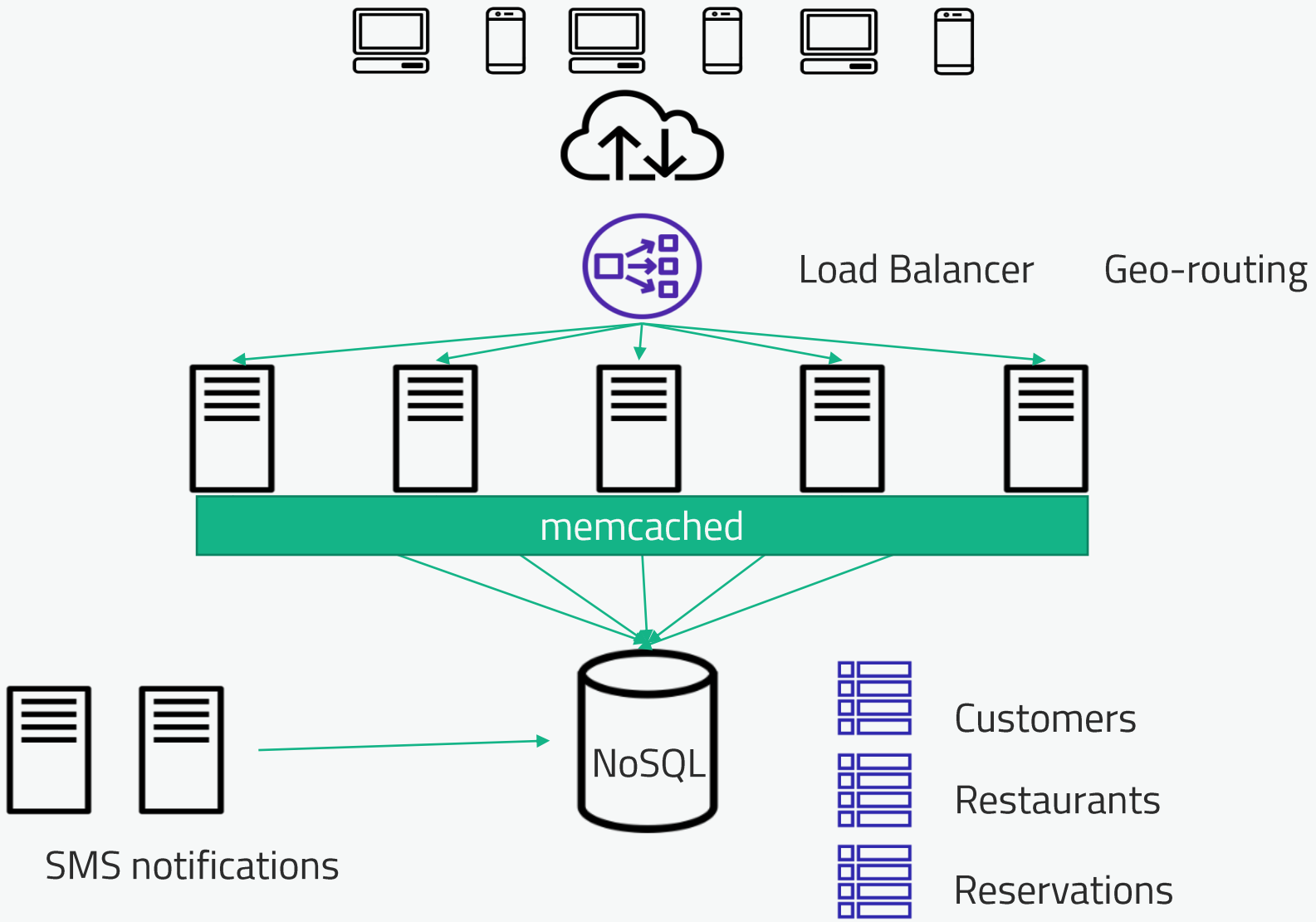
- How would you organize the data needed for this system?
- How would you design a system that reliably scales to thousands of restaurants and hundreds of thousands of users?

Let me sketch some thoughts on the data we'll need while I'm thinking of it…

Reservation
ID
Customer ID
Restaurant ID
Time slot
Party size
Notes (special occasion, dietary restrictions, etc.)

Restaurant
ID
Table layout (# tables, seats per table)
Walk-up holdback
Reservation length
Business hours
Name
Address
Phone#

Customer
ID
Primary contact name
Phone #
Email
Preferences
Location


Load Balancer
Geo-routing
memcached
NoSQL
SMS notifications
Customers
Restaurants
Reservations

DEBRIEF

- Started by clarifying requirements and the scale of the system
- Worked backwards from the customer experience
- Thought through the data needed and how it relates to each other, and how to efficiently store it
- Proposed a horizontally scalable fleet of app servers, distributed to maximize availability
- Proposed an appropriate distributed database
- Did not get defensive when challenged by the interviewer
- Made the design better within the time available (with addition of caching)


DESIGN A WEB
CRAWLER

We’re designing a web crawler. Like, the entire web – or just a few sites?

I thought you might say that. So we’re talking, like, billions of web pages. Crawled how often?

And, we need to check pages we’ve crawled before to see if they have been updated, right?

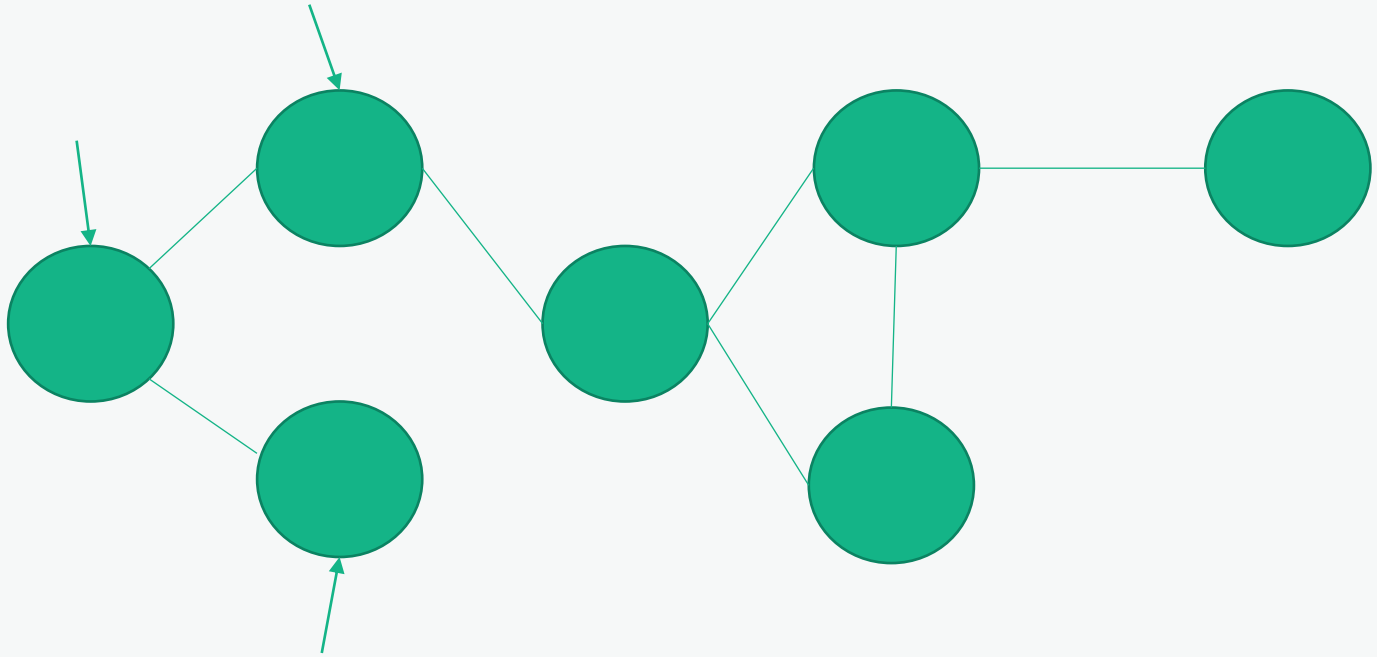
OK, do we need to store a copy of every page as we go? Does that include images?

What about dynamic content? Stuff that's rendered client-side?

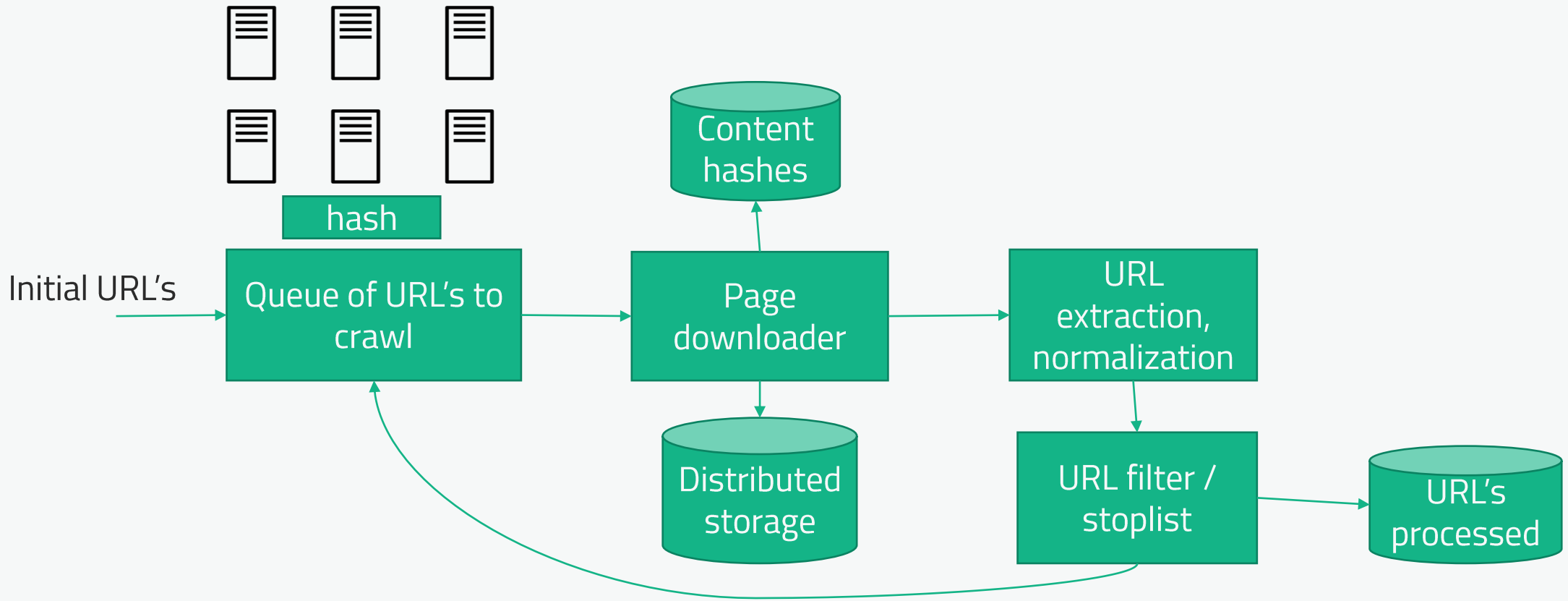
What's the main purpose of this crawler? I should've asked that first really.

Try It Yourself!

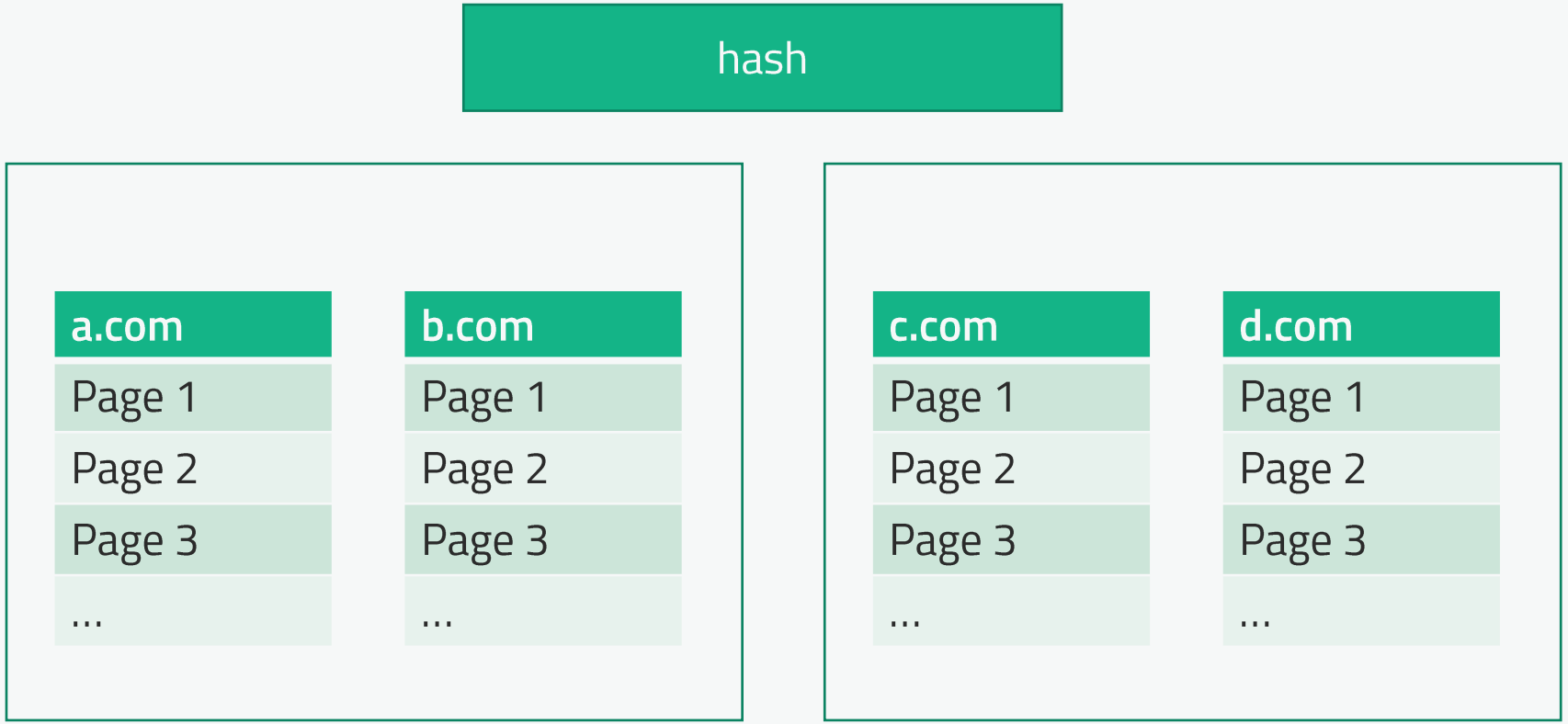
- How would you distribute this crawler to handle the massive scale required?
- What algorithm(s) will you use to crawl the web?
- What problems and failure modes can you anticipate and address?

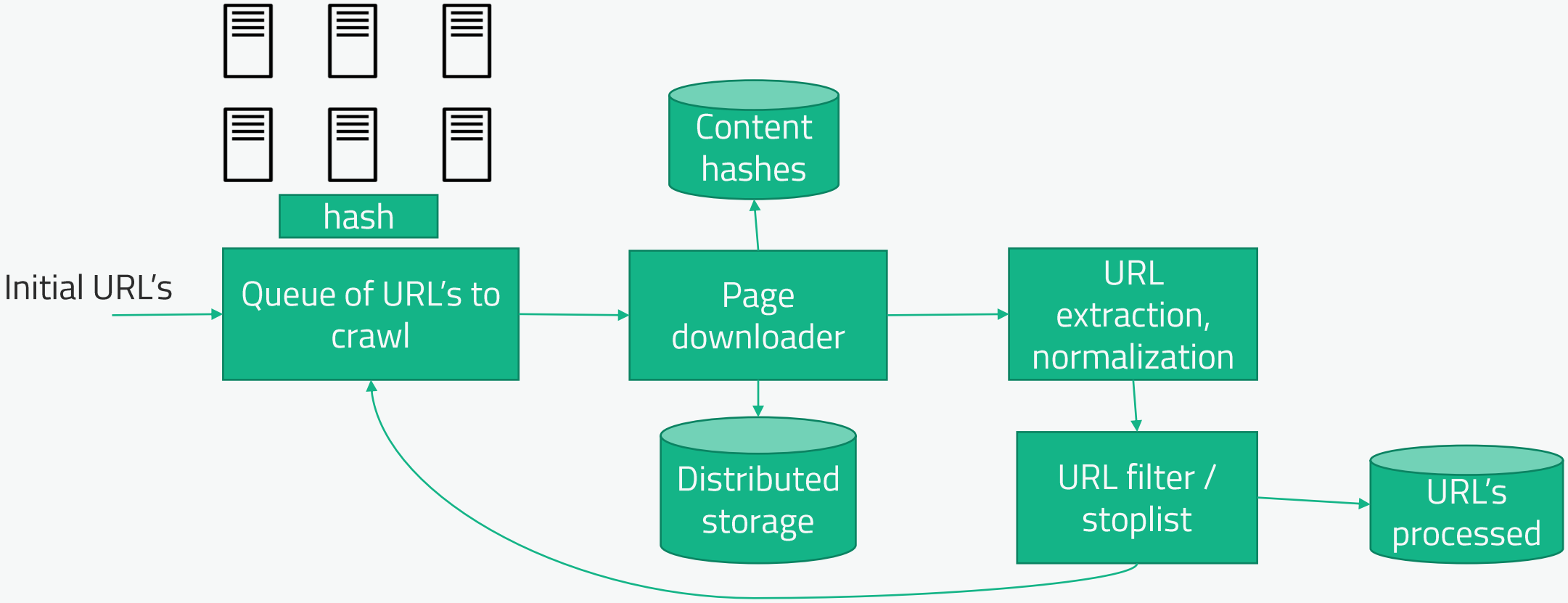


BFS


Content hashes
hash
Initial URL's
Queue of URL's to crawl
Page downloader
URL extraction, normalization
Distributed storage
URL filter / stoplist
URL's processed

Page downloader




Content hashes
hash
Initial URL's
Queue of URL's to crawl
Page downloader
URL extraction, normalization
Distributed storage
URL filter / stoplist
URL's processed

DEBRIEF

- Started by clarifying requirements and the scale of the system
- This isn't a customer-facing system, so we just worked backward from those requirements.
- We started with a high level design, and refined it as time permitted.
- We demonstrated knowledge of data structures and algorithms and how to apply them
- We addressed ways to scale things that don't have out of the box solutions
- We struck a collaborative tone with the interviewer when working through issues with the design
- We demonstrated a desire for simplicity

This is not the only solution!

- The interviewer is not concerned with you getting exactly the same architecture they are using in the real world.
- They just want to see your thought process. So think out loud, sketch on the whiteboard, let them see how you are attacking the problem.
- This gives the interviewer the opportunity to steer you in the right direction as well (or away from the wrong one.) Partly they just want to see how you take criticism and feedback.
- It's the tools and design patterns that are important, not this specific architecture.

Design a Top-Sellers feature for an ecommerce website.

OK, we're designing the system that computes top-sellers. Is this top-sellers across the entire site, or broken down by category?

How up-to-date do the results need to be?

Right, over what period of time are we computing top-sellers?

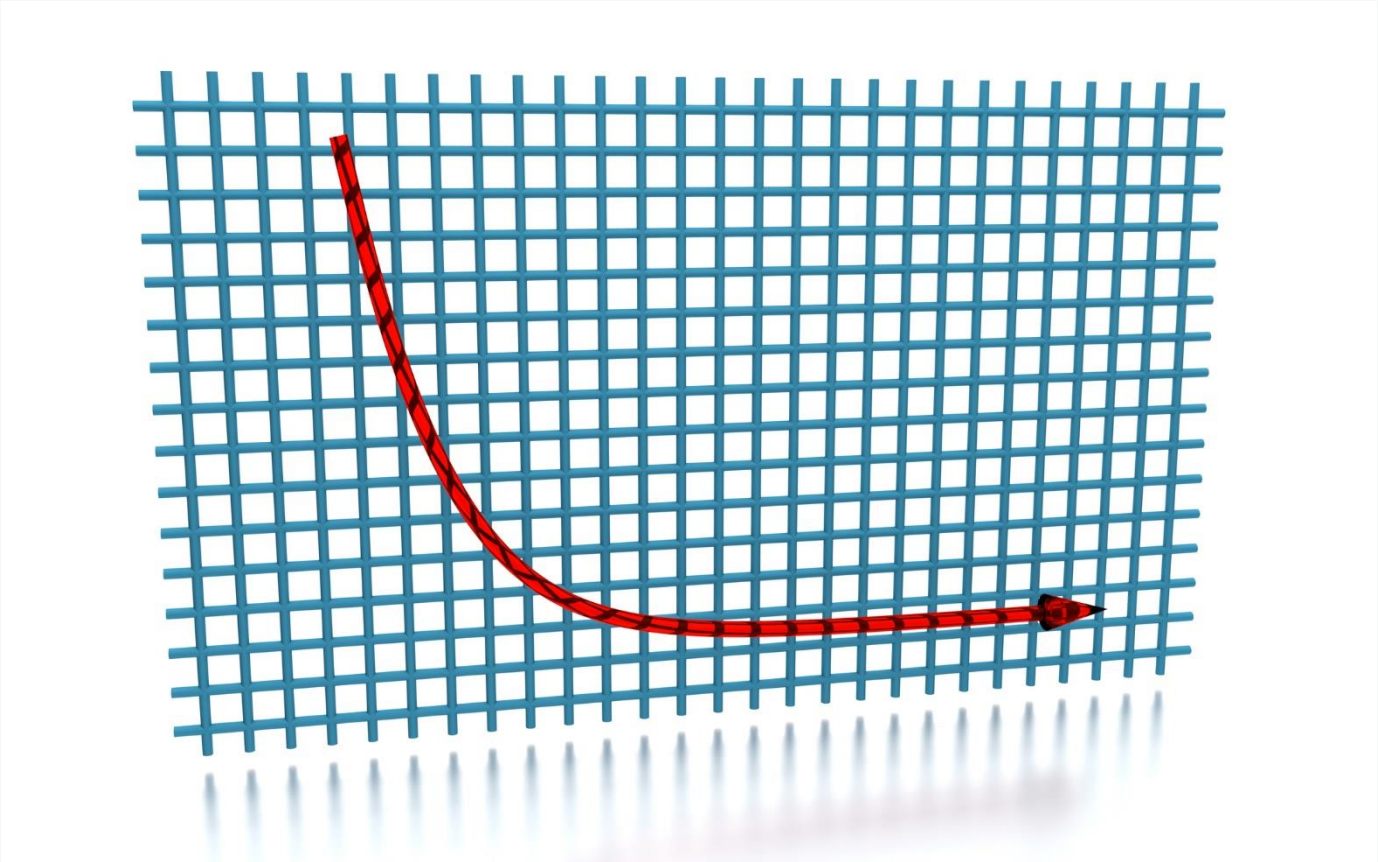
Maybe we just look at all sales, but give them less weight over time? So older purchases count less than new ones.

What sort of scale are we looking at here?

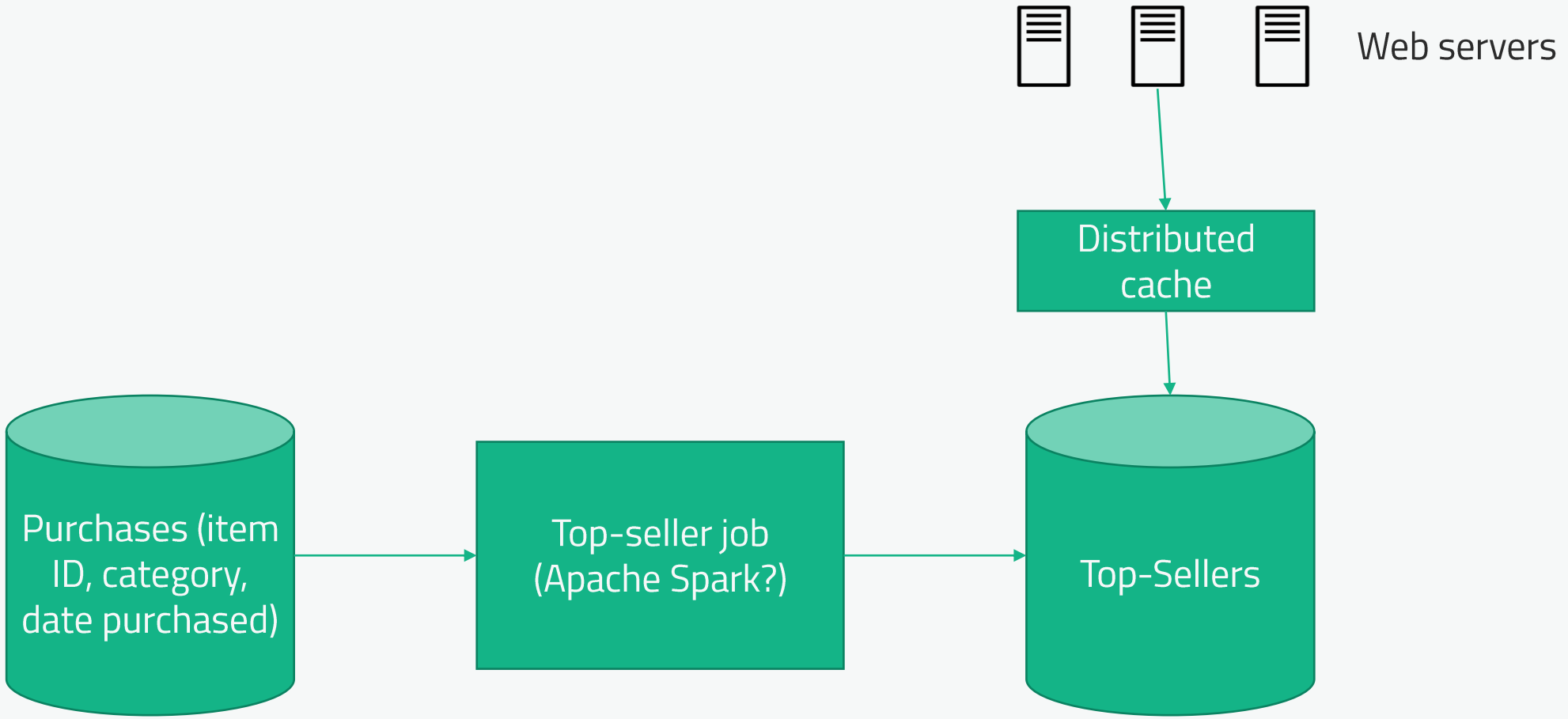
Try It Yourself!

- You've actually gotten through most of the test already on this one… when I used this question at Amazon, I was really using it to see if people could see things from the customer's perspective and anticipate these sorts of issues before they even started designing.

- But you've gotten past that! What sort of system design would fulfill the top-sellers feature we've discussed?



$$e^{-\lambda t}$$


Web servers
Distributed cache
Purchases (item ID, category, date purchased)
Top-seller job (Apache Spark?)
Top-Sellers

DEBRIEF

- Started by clarifying requirements and the scale of the system
- Asking the right questions was half of the interview! The design part was relatively simple.
- Thinking about what customers expected to see was the key.
- Having a good toolchest available helped with the design (S3 data lake, Apache Spark, DynamoDB for example.)
- We faced the cold start problem for caches.
- We struck a collaborative tone with the interviewer when working through issues with the design
- We demonstrated a desire for simplicity

DESIGN A VIDEO SHARING SERVICE

I need to design a video sharing service. So, we're talking something like YouTube?

YouTube has a lot of features… recommendations, channels, advertising… it's not just storing and playing back videos. What features should I focus on?

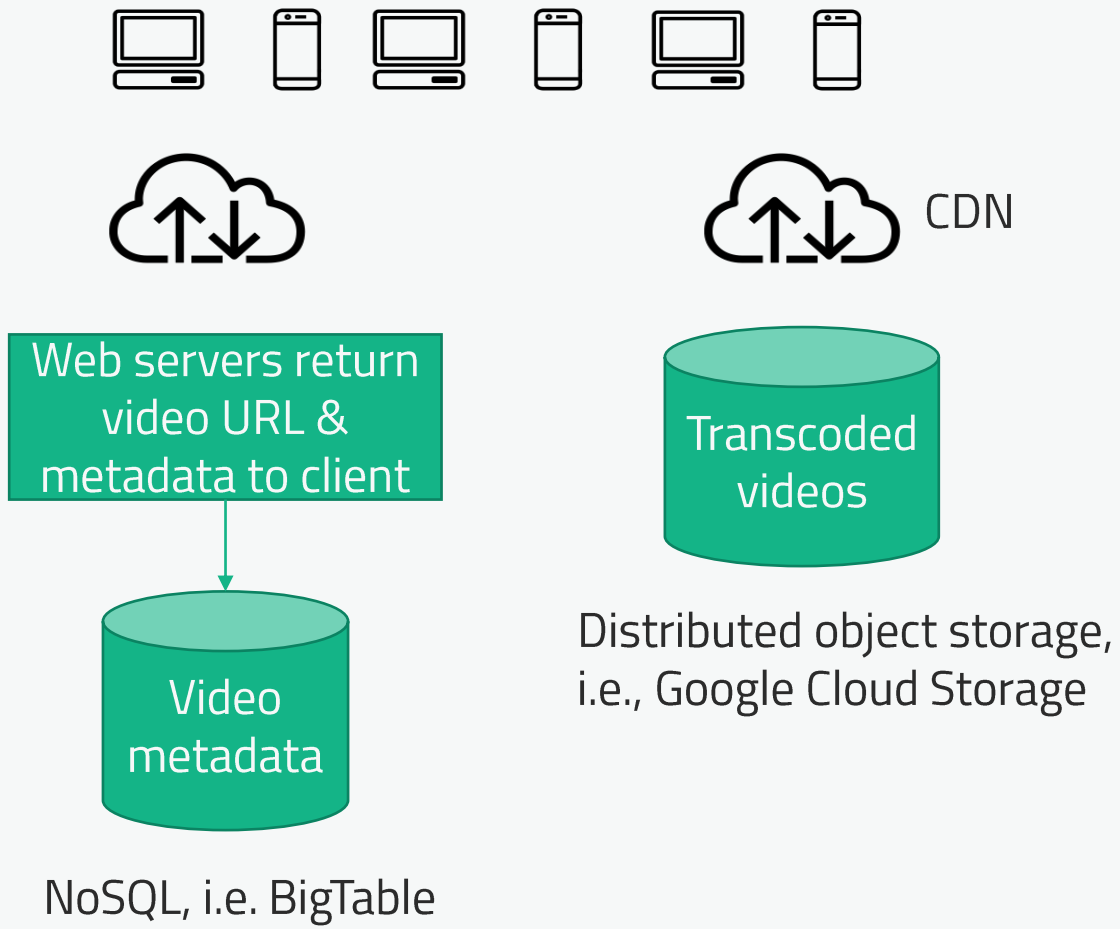
So, we're talking about users and videos in the billions, and people uploading and watching all around the world, right?

Alright, so it seems like there are two things I need to cover: handling video uploads, and handling video playback. At massive scale. Sounds right?

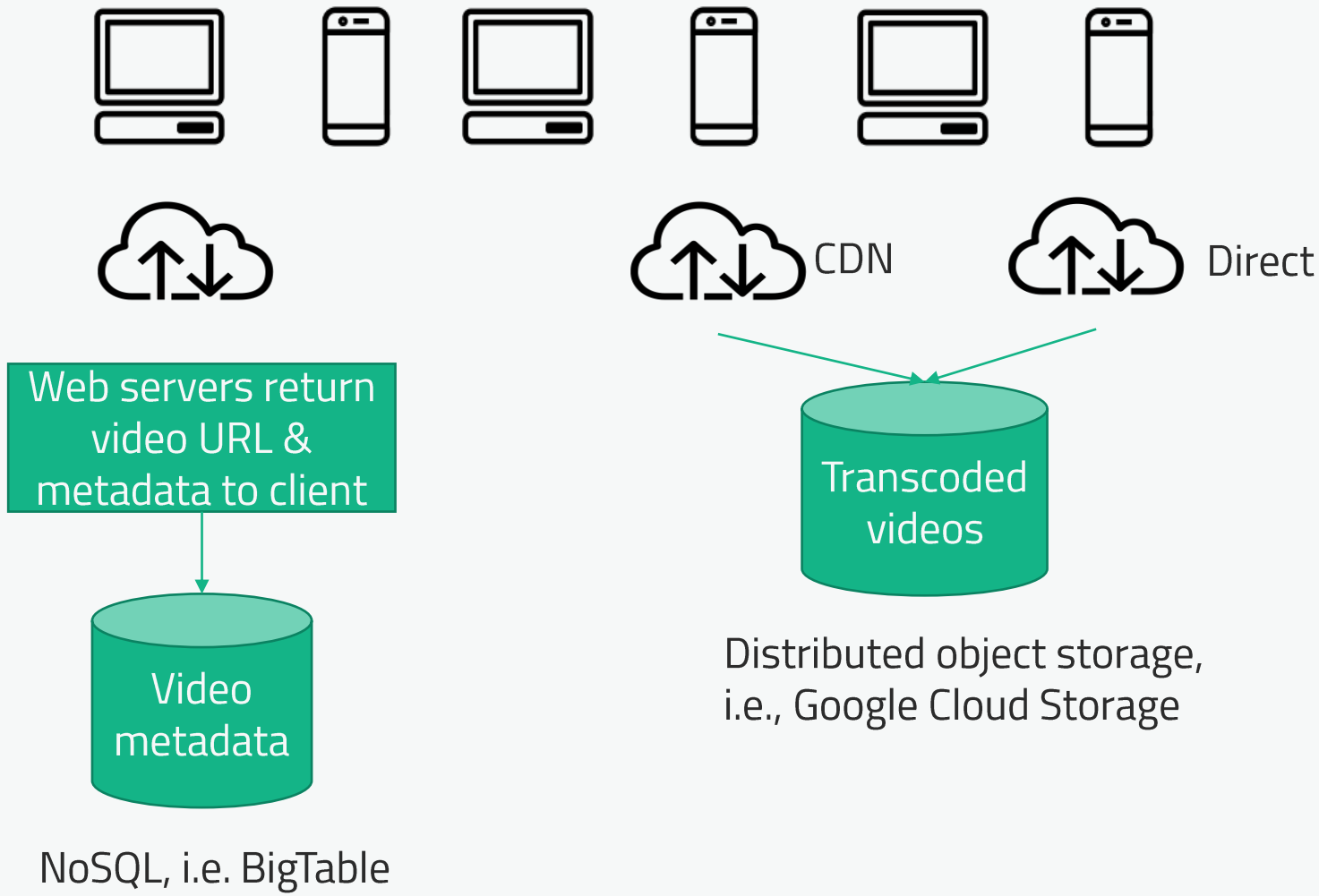
Try It Yourself!

- How would you design a system to allow users to upload, transcode, and vend videos around the world efficiently?
- Having a good toolchest at your disposal is key; you aren't expected to develop the various sub-components from scratch if there are cloud solutions available.
- But, do think about the cost of these services and how you might keep those costs down.

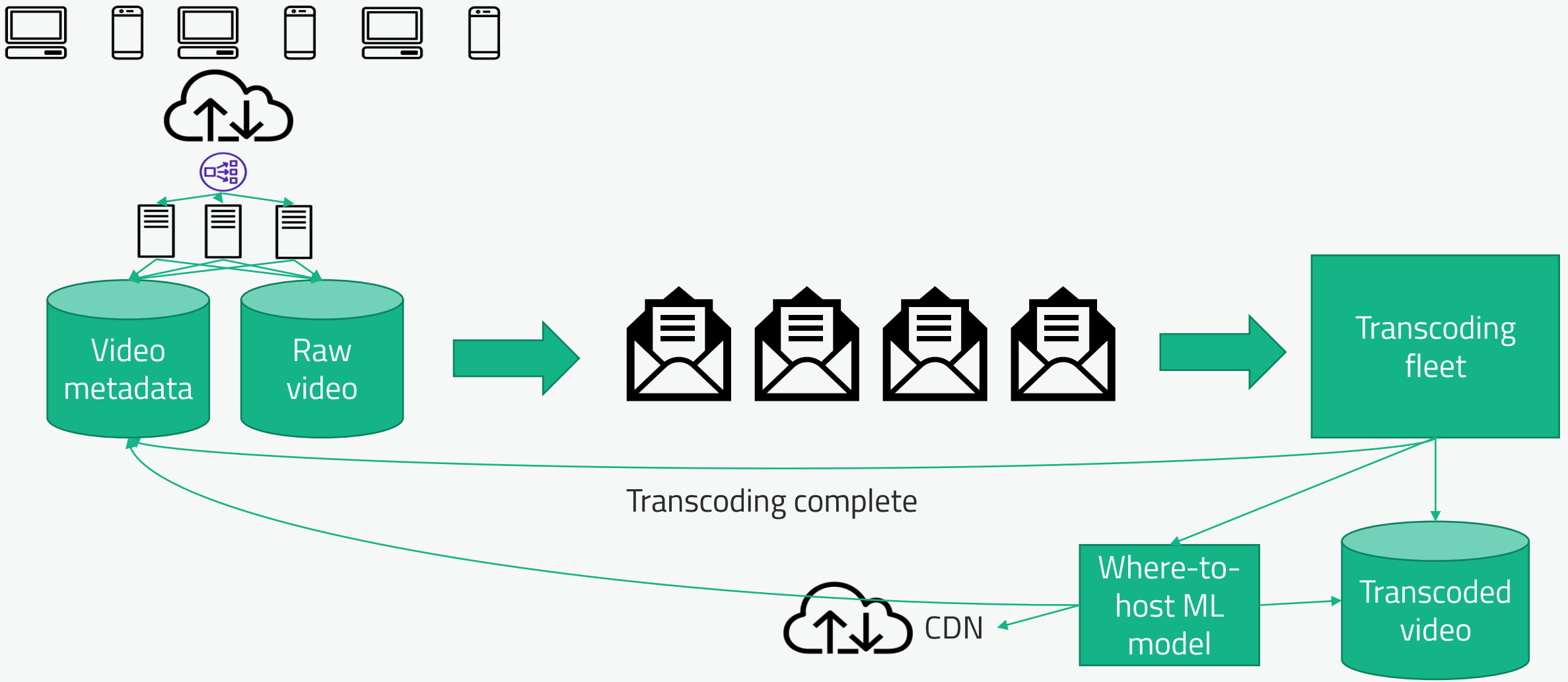
Video playback



Video playback



Video uploading



DEBRIEF

- The test here was if you could handle a question as open-ended as “design YouTube” and break it down into manageable pieces.
- You had to start with questions as to what the interviewer wanted you to focus on.
- Again in our designs, we always started with the client / customer and worked backward.
- We kept things high-level at first, to ensure we had a cohesive design in the time we had. We didn’t get into how BigTable works etc.
- We proposed technical solutions to business problems (controlling CDN costs)
- We got to apply message queues, NoSQL, stateless servers, CDN’s, and even machine learning in our solution.
- Again there are many ways to approach this problem; this is only one.

Design a Search Engine

Are we talking about a search engine for the entire web, like Google, or just some intranet tool?

OK. We designed a web crawler earlier, so I can assume we have a distributed data store of page content already, right?

I'm not going to recreate all of Google in 20 minutes. Can we focus just on the problem of generating reasonable search results at massive scale?

Try It Yourself!

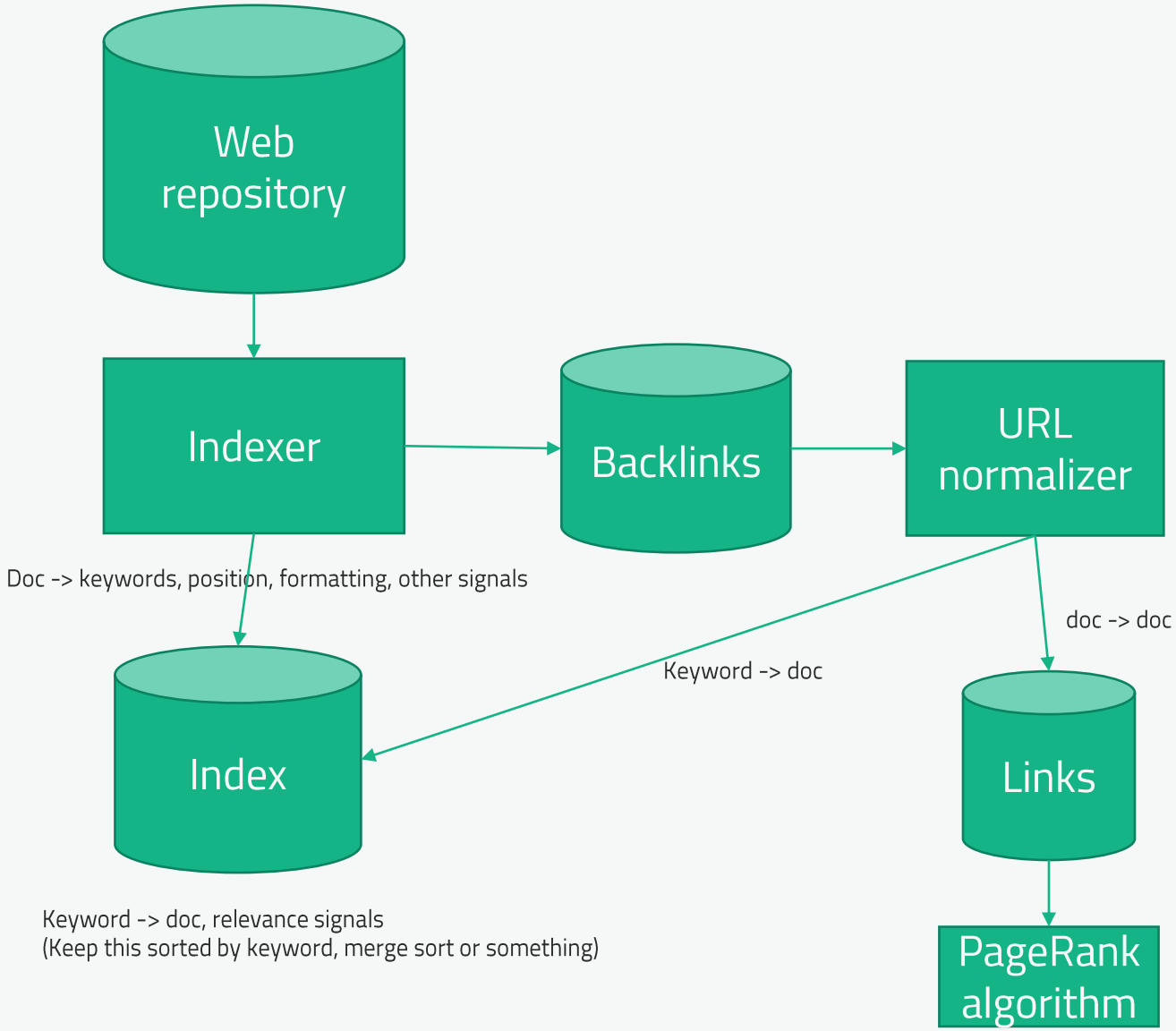
- What data would you want to extract to measure how relevant a page is to the keywords within it?
- What algorithms would you use to map keywords to pages, and sort them?
- What system architecture would allow you to do all this at ludicrous scale?

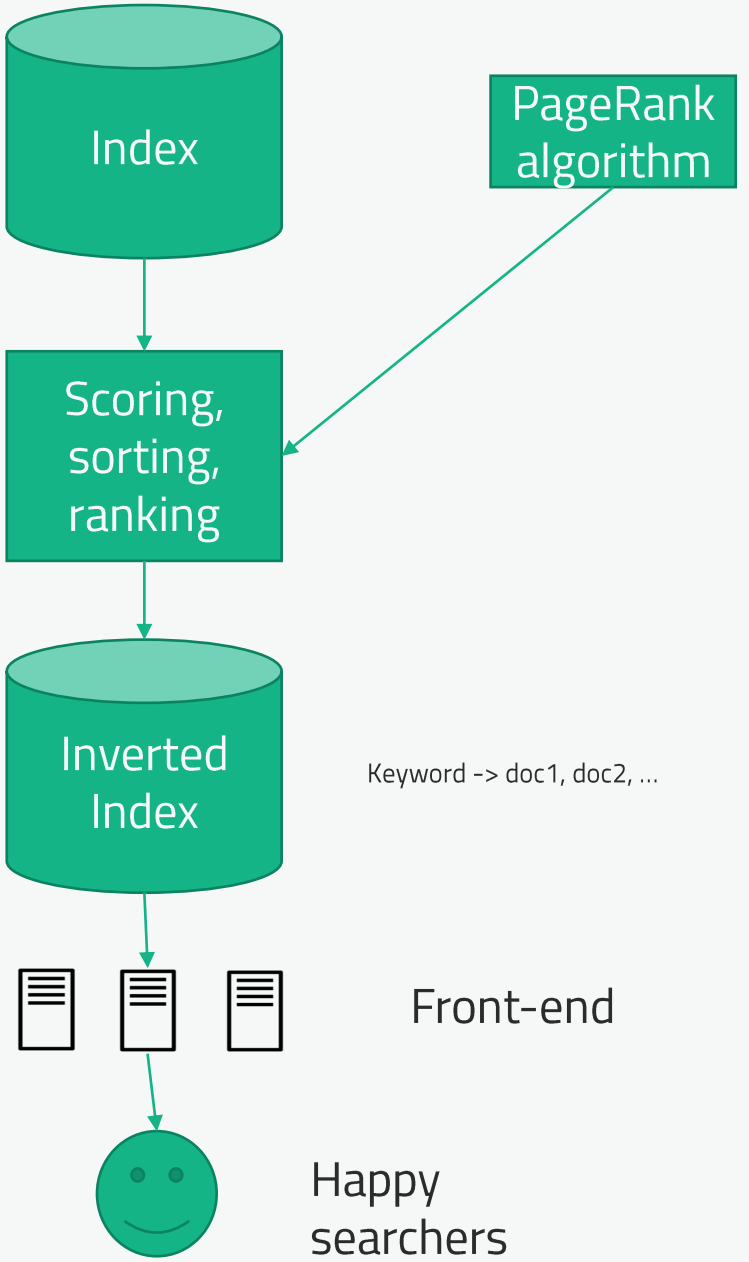
$$\frac{Term\ frequency}{document\ frequency}$$

Inverted index

Keyword -> sorted list of documents

Backlinks
Terms in the doc
Their position
Font size / headings / etc. (formatting)
Titles
Length of document
Term frequency
Metadata


Web repository
Indexer
Backlinks
URL normalizer
Doc -> keywords, position, formatting, other signals
Keyword -> doc
doc -> doc
Index
Links
Keyword -> doc, relevance signals
(Keep this sorted by keyword, merge sort or something)
PageRank algorithm


Index
PageRank algorithm
Scoring, sorting, ranking
Inverted Index
Keyword -> doc1, doc2, ...
Front-end
Happy searchers

DEBRIEF

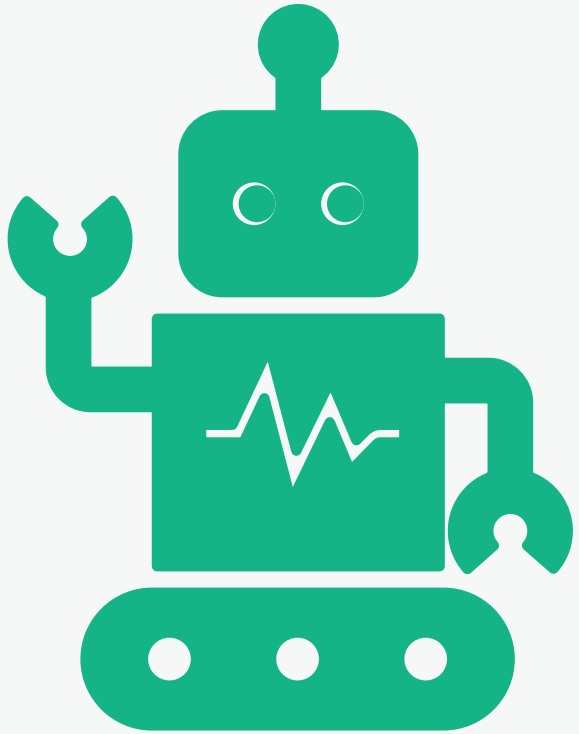
- This is a very complex problem, and if you're not already familiar with how Google was originally designed, you're being asked to be as smart as Sergey Brin and Larry Page here.
- Although we did not work backwards in the design strictly speaking, we did start with the knowledge that we wanted an inverted index in the end, and had a web page repository in the beginning, and we had to figure out how to get between the two.
- Given the complexity we focused on high-level system architecture and did not go into detail on any given sub-component unless time later allowed.
- Honesty is critical. It's OK to say you don't know the details of PageRank or how Google works today, but you should acknowledge that and what problems you are leaving unsolved.
- This is not the only solution nor the best one. But that's not important; what was important was how you reacted to questions and feedback from the interviewer, and that you "thought out loud" to give the interviewer an opportunity to steer you in the right direction.
- Listen to hints from the interviewer; we quickly abandoned TF/IDF since we were subtly being steered away from it.

General Tech Interview Tips

What are hiring managers really looking for?

- Determination.
- Grit.
- Perseverance.
- Whatever you want to call it.

Technology changes quickly. Your determination to quickly learn that technology does not.



Yes, you STILL NEED TO PROVE YOU CAN CODE. On a whiteboard or digital equivalent.

Tech skills matter, but they are just table stakes.

- How do you demonstrate perseverance?

Tell them a story.

Hiring managers practice "behavioral interviewing"
They want to know how you have reacted to specific challenges in the past.
Come prepared with STORIES about how YOU solved challenging problems on your own. The interviewer will dig into details.

Be Ready for...

As you go through the interview process, you'll be interviewed by engineers, architects, managers, and someone like me.

- **Technical Skills**
  Coding-at-the-whiteboard; system design problems.
- **Your Experience**
  STORIES about tough problems you had to solve; be prepared to dive deep.
- **Your Fit with Company Values**
  Research what they are, and have STORIES ready to demonstrate you possess them.

What They Want

These are signs of the perseverance hiring managers seek.

- **Independent Thought**
  Can you research solutions to new problems on your own?
- **Independent Learning**
  When faced with a new technology, can you quickly learn it on your own? (Hey, Udemy can help!)
- **Never Give Up, Never Surrender**
  Do you have the grit to see challenging problems through to completion?

You must be self-motivated.

You shouldn't need to be told that watching cat videos all day because your boss didn't give you specific instructions is not OK.

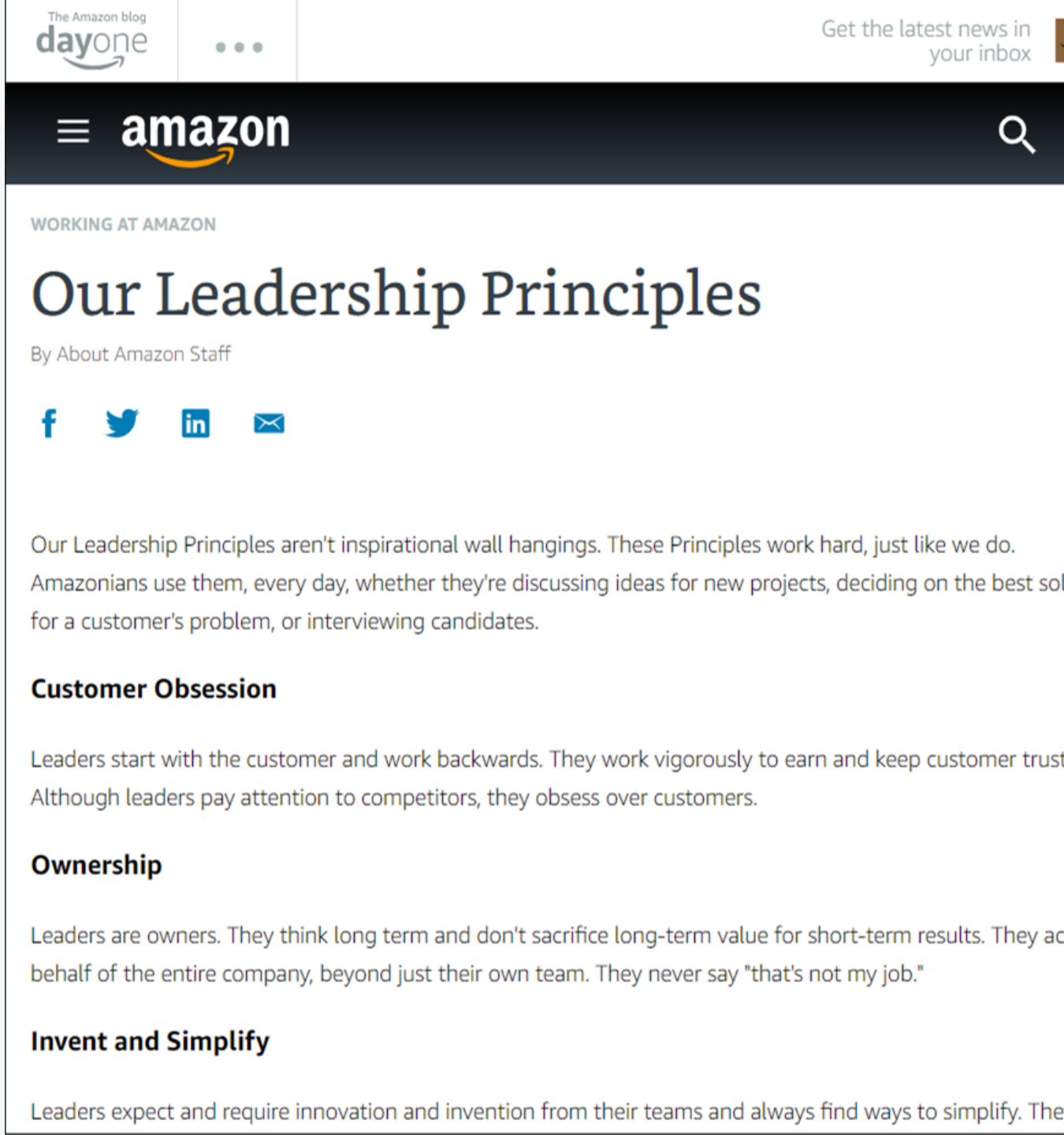
Have stories of your INITIATIVE. Did you take on new work on your own, or develop an idea of your own, in your spare time? Hiring managers LOVE that.

What They Don't Want

- **Let Me Google That For You**
  People who constantly lean on others for basic guidance won't last long.
- **Step by Step Instructions**
  If you can't accomplish anything without a recipe, you can't solve the new and unique problems your employer faces.
- **Failure of Focus**
  You must appreciate that your work has ZERO value until it is deployed and in front of customers.

Understand the Company

- LEARN THE COMPANY VALUES
- i.e., Amazon's leadership values / customer focus, Google's "ten things"
- Demonstrate these values in the stories you tell!



Practice Coding and Designing at the Whiteboard

Writing code while someone is watching you takes some getting used to.

There are plenty of sample coding exercises out there to practice with.

•Bring your Stamina

•Try to arrange your travel schedule so you'll have time to acclimatize and rest.

•Don't show up tired. Exercise, drink some energetic drink thing, whatever.

•Eat breakfast! And use the bathroom before heading in.

Please take a shower.

I really shouldn't have to say this.

But I do.

Think about your questions for them.

•Nothing's worse than saying "um, nope" when an interviewer says "do you have any questions for me?"

•Display some curiosity. Ask about their typical day at the company. Ask about how career progression works. Ask about *their* biggest challenge.

Think Big.

Think about larger systems, running at massive scale. Any system you design must hold up to petabytes of data / thousands of transactions per second.

Think about the business, and not just the technology.

Be nice.

- They're not only evaluating your technical skills. They're evaluating what it's like to work with you.

- Smile. Ask about their jobs. Stay positive. Stay humble.

Do your research.

As a hiring manager, I hated websites that collected interview questions from specific companies.

But you should love them.